TECHNIQUES TO SPEED-UP COUNTING BASED DATA MINING ALGORITHMS ON GPUS

Amila De Silva

168062J

Degree of Master of Science

Department of Computer Science & Engineering

University of Moratuwa
Sri Lanka

April 2019

TECHNIQUES TO SPEED-UP COUNTING BASED DATA MINING ALGORITHMS ON GPUS

Amila De Silva

168062J

Thesis/Dissertation submitted in partial fulfillment of the requirements for the degree Master of Science in Computer Science and Engineering

Department of Computer Science & Engineering

University of Moratuwa

Sri Lanka

April 2019

DECLARATION

I declare that this is my own work and this dissertation does not incorporate without acknowledgement any material previously submitted for a Degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant to University of Moratuwa the non-exclusive right to reproduce and distribute my dissertation, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works (such as articles or books).

Signature: Date:

The above candidate has carried out research for the Masters thesis/Dissertation under my supervision.

Signature of the Supervisor: Date:

ACKNOWLEDGEMENTS

I am sincerely grateful for the advice and guidance of my supervisors Dr. Shehan Perera and Prof. Sanath Jayasena. Without their help and encouragement this project would not have been completed. I would like to thank them for taking time out of their busy schedule to be available anytime that was needed with help and advice.

I would also like to thank my progress review committee, Dr. Surangika Ranathunga and Dr. Lochandaka Ranathunga. Their valuable insights and guidance helped me immensely.

I would like to thank the entire staff of the Department of Computer Science and Engineering, Both academic and non-academic for all their help during the course of this work and for providing me with the resources necessary to conduct my research.

My sincere gratitude goes to Senate Research Grant for funding this study. This work was partially funded by the LK Domain Registry through Prof. V.K. Samaranayake top-up grant.

Finally, I would like to express my gratitude to my family and all my friends for their support.

ABSTRACT

Techniques to speed-up counting based Data Mining Algorithms on GPUs

Data Mining by its definition is meant to deal with large volumes of data. Ever growing volumes of Data and increasing demand for data driven decisions are placing new requirements on Data Mining algorithms. To respond to these demands Data Mining practitioners are focusing on improving speed and turnaround time without compromising accuracy.

Among different approaches in improving speed, one approach gaining increased attention is the use of GPUs. Ability of GPUs to perform parallel executions at a massive scale and inherently repetitive nature of Data Mining workloads make GPUs a better candidate in improving speed.

Another area getting increased attention is using Bitmaps for Data Mining algorithms. Bitmap representations have been abundantly used in analytical queries for their ability to represent data concisely and for being able to simplify processing.

A number of studies have been carried out which combine these two techniques to achieve greater performance improvements. But most of those studies are revolving around FIM based algorithms, processing of which naturally aligns with Bitmap representations.

In this study, we explore the ability of using Bitmap techniques on GPUs to speed up a class of Data Mining Algorithms. A Counting based Algorithm can be defined as an Algorithm which can be separated into to two distinct phases a pattern counting phase and a model building phase. We propose a framework based on Bitmap techniques, which speeds up these counting based algorithms on GPUs. The proposed framework uses both CPU and GPU for the algorithm execution, where the core computing is delegated to GPU. We implement two algorithms Naïve Bayes and Decision Trees, using the framework, both of which outperform CPU counterparts by several orders of magnitude.

Keywords: Data Mining; GPU; Classification; Bitmaps;BitSlices;Naïve Bayes;Decision Trees

LIST OF FIGURES

Figure 3.1	Data set represented using Bitmaps.	17
Figure 3.2	Data set represented using Bit-Slices.	17
Figure 3.3	Converting a single column to Bit-Slices.	19
Figure 3.4	Counting occurrences of a number with a Bit-Slice column.	20
Figure 3.5	Converting two columns to Bit-Slice representation.	20
Figure 3.6	Counting co-occurrences with Bit-Slices.	21
Figure 3.7	Bitmap intersection & counting on GPU.	24
Figure 3.8	Bit-Slice intersection & counting on GPU.	28
Figure 3.9	Building Decision Trees with Bitmaps.	33
Figure 4.1	Execution time vs no. of instances - CPU Algorithms	39
Figure 4.2	Execution time vs no. of instances - CPU and GPU Algorithms	39
Figure 4.3	Execution time vs no. of instances - GPU Algorithms	40
Figure 4.4	Execution time vs no. of Patterns	40
Figure 4.5	Execution times with Different Data sets Results for Naïve Bayes.	41
Figure 4.6	Executions on GPU with the three Data Sets.	42
Figure 4.7	Speedup over Standard-CPU on different Data Sets.	43
Figure 4.8	Naïve Bayes speedup vs instance count	44
Figure 4.9	Execution times for Decision Tree with Different Data Sets	45
Figure 4.10	Speedup over Standard-CPU on different Data Sets.	46

LIST OF TABLES

Table 4.1 Different implementations and their descriptions. 41

LIST OF ABBREVIATIONS

Abbreviation	Description
ARM	Associate Rule Mining
FIM	Frequent Itemset Mining
GPGPU	General purpose Graphic Processing Units
IOT	Internet of Things
SIMD	Single Instruction Multiple Data

TABLE OF CONTENTS

Declaration of the Candidate & Supervisor i
Acknowledgement ii
Abstract iii
List of Figures iv
List of Tables v
List of Abbreviations vi
Table of Contents vii
1 Introduction 1
1.1 Data Mining on GPUs 1
1.2 Using Bitmap techniques for Data Mining 2
1.3 Problem Statement 3
1.4 Our Solution 4
1.5 Contributions 4
1.6 Organization 5
2 Literature Survey 6
2.1 Using GPUs for General purpose computing 6
2.2 Parallel Data Mining Algorithms that uses Bitmaps 7
2.2.1 Two FIM implementations with Bitmaps 7
2.2.2 gpuDCI 8
2.2.3 GMiner 9
2.3 GPU Frameworks for Data Mining Applications 9
2.3.1 GPUMiner 9
2.3.2 Index Structure for Similarity Joins 10
2.3.3 Framework for Mapping Data Mining Applications on GPUs 11
2.3.4 Three techniques to improve Data Mining algorithms 12
2.4 GPU Implementations for Naïve Bayes and Decision Trees 12
3 Methodology 14
3.1 Architecture 14

3.2 Bit-Slice And Bitmap Representations 16
3.3 Processing Bit-Slices on a CPU 18
3.3.1 Counting a single element with Bit-Slices 18
3.3.2 Counting co-occurrence of two elements 19
3.4 Processing Bitmaps And Bit-Slices on GPU 23
3.4.1 Processing Bitmaps on GPU 23
3.4.2 Bit-Slice Processing on GPU 26
3.4.3 Batching operations 28
3.5 Algorithm Execution 29
3.5.1 Implementing Naïve Bayes on framework 29
3.5.2 Implementing Naïve Bayes 30
3.5.3 Implementing Decision Trees on framework 31
3.5.4 Building Decision Trees with Bitmaps 32
3.5.5 Running Decision Trees on GPU 34
4 Experimental Results 36
4.0.1 Data Sets 37
4.0.2 Experimental setup 38
4.0.3 Results for Co-OccurrenceCount 38
4.0.4 Running time for Naïve Bayes 39
4.0.5 Results for Decision Trees 44
5 Conclusions and Recommendations 47
References 49

Chapter 1

INTRODUCTION

This chapter gives an overview on Data mining and briefly explains challenges faced with analysing large volumes of data and methods being followed currently to overcome those. This chapter also details the solution we are proposing and highlights our contribution. The latter sections explain how the thesis is organised.

1.1 Data Mining on GPUs

Data Mining is defined as the process of finding patterns in Data according to [1]. The process usually involves scanning through large volumes of Data gathered through disparate systems. The mining process is often an automated process where data is ingested from different systems, refined and presented to different algorithms where the actual Pattern Extraction Happens. According to Forbes [2] proliferation of IOT devices and presence of internet have had such an impact on data generation that nearly 90 percent of data has been generated over the last 2 years. To handle such growing volumes, researchers are on the look for efficient Data mining tools.

Distributed Data processing frameworks such as Hadoop [3] and Spark [4] emerged as a solution to this problem. These frameworks are capable of utilising commodity hardware to scale up processing which helps bringing down processing time. Amidst the presence of distributed Mining solutions, quest for efficient Data mining solutions has not stopped. Distributed frameworks do allow churning data in a parallel manner, but to achieve a higher degree of parallelism, Data and commands need to be transferred over the network, which sometimes can become a bottleneck. Since these grids rely on commodity hardware, processing nodes do not increase grid resources in a proportional manner. There can be chances when a more computing power is needed, but all offered from an additional node

is additional Memory and storage. These irregularities are motivating researches to find options which allows scaling vertically first before going horizontally.

It is in this light researchers are turning their attention to GPUs. With the availability of GPGPUs researchers are trying to achieve better scalability on a single machine. Use of GPUs in Data Mining is not totally new. At a time when GPUs have been first evaluated for General purpose computing, studies have been carried out on using GPGPUs for database operations. And later those studies have evolved into writing fully fledged Data mining algorithms on GPUs. Data Mining algorithms can be easily ported onto GPUs because SIMD parallelism offered by those naturally aligns with the processing that happens in Data Mining algorithms. In a typical Data Mining algorithm, same operations needs to be executed for different data points.

In most of the instances where GPUs have been used, they have been used to accelerate a single algorithm. Some studies focus on running the entire algorithm on the GPU, while some others try to use a CPU-GPU hybrid approach. Apart from one or two instances, most of the studies are creating specific data structures and data processing models, only applicable to one specific algorithm. There are few exception to these, which try at providing a common framework to be used by different algorithms. The few studies we went through either provide support for basic operations, such as data transferring [5] or specifically improves one core algorithm and improves supporting operations for other algorithms [6]. We in our study attempt at coming up with a generic processing framework, where algorithm execution happens in two phases. A phase in which Data set is queried to count patterns and a phase in which algorithm is built to use the obtained counts.

1.2 Using Bitmap techniques for Data Mining

As early as 1997 Bitmap based techniques have been used when evaluating long predicate statements. Different varieties of Bitmap indices have been proposed to obtain aggregates. Bitmaps are simple structures which can be stored with less

space which enables those to be loaded into memory quickly. Bitmap intersecting is a simple operation that can be performed in an efficient manner with the bit-wise operations available in CPUs. These reasons in combination have helped in speeding up analytical queries when Bitmaps are used. The organisation and the processing done with Bitmaps makes it a natural candidate for FIM algorithms. In FIM algorithms such as Apriori, individual itemsets are mapped into distinct Bitmaps so that generation of new itemsets can be easily done by intersecting Bitmaps. Due to this, Bitmaps are widely used as a popular data structure in FIM algorithms. But before being used in FIM algorithms, Bitmaps have been considered as a complementary index, not as a complete data structure. Simplicity in processing and lower consumption of memory makes Bitmaps a good fit for processing on GPUs. Bitmap processing allows multiple GPU cores to be used in parallel, allowing a higher degree of parallelism. Researchers have often obtained speed ups by several orders of magnitude when using Bitmaps. For these reasons, Bitmaps have been used in several GPU based Data Mining frameworks. The most basic operation done with Bitmaps is intersecting and counting number of 1s. Since this is the same core operation performed in FIM algorithms, Bitmaps are mostly used for FIM implementations. But this processing does not need to be limited to FIM and can be extended to other Algorithms like Naïve Bayes. One of the objectives of this study is to evaluate the extent to which Bitmaps can be used for other algorithms through a common framework.

1.3 Problem Statement

Currently Bitmap techniques are being used with GPUs to speed up Data Mining algorithms. But in most of the cases either the use of Bitmaps is very specific to the algorithm, or Bitmaps are only used as a complementary structure which only gets used for minor processing. So far FIM algorithms are using Bitmaps as a core Data Structure, which helps those algorithms to achieve great speeds. Lack of a generic framework limits the benefits offered by Bitmaps to FIM algorithms.

1.4 Our Solution

We propose a Data Structure based on Bitmaps which will act as the main in-memory storage. This structure will be loaded onto the GPU during model building. We also provide a set of core algorithms that would query the Data set for a given pattern and would output the number of occurrences. We provide a framework comprised with these features, which can be used to implement a High Level data mining algorithm by expressing those as a series of core pattern counting operations. The framework supports two Bitmap variations Bit-Slices and Bitmaps each being good at a particular type of operation. We are using Bitmaps for the underlying storage, because

- It provides parallel elements on which different processing units can works on simultaneously
- Memory consumption is low which allows more elements to be loaded into memory and reduces frequent transfers between Host and Device
- Bitmaps can be represented with Arrays and GPU kernels can be efficiently written to process arrays.

1.5 Contributions

Our main contribution is a framework based on Bitmap Techniques for Data mining Algorithms. In our study we propose a uniform approach for querying data, which is algorithm agnostic. The novel contribution we make is the Batching technique, which can be applied on many Bitmap processing algorithms. Since this help improving speed, we have used batching for two main operations. We also provide implementations for two algorithms, Naïve Bayes and Decision Trees using Bitmap Techniques. By expressing these algorithms in counting operations, we propose a method to use Bitmap manipulations. As a summary we can list our contribution as below;

- A framework based on Bitmap Techniques for data mining algorithms.

- Batching technique to improve speed of Bitmap based algorithms.
- Implementations for Naïve Bayes and Classification Trees using Bitmap Techniques

1.6 Organization

The rest of this document is organized as follows. Chapter 2 presents the past work done in the area of Data Mining, Bitmaps and specifically on GPU based Frameworks for Data Mining. Chapter 3 presents the methodology we used in proposing our framework. Chapter 4 describes the experiments we performed with the framework. Chapter 5 discusses the results and related areas we can further research on.

Chapter 2

LITERATURE SURVEY

In this section, we take a look at previous work done on implementing Data Mining algorithms with GPUs. We start with a short review on using GPUs for computational tasks, where we review different approaches followed while measuring performance. Then we review work done on Data Mining Algorithms that use Bitmaps. Under this section we review different implementations done for FIM algorithms. Next we look at GPU based frameworks proposed for Data Mining algorithms. Since we implement Naïve Bayes and Decision Trees, in the final section we review work done on parallelizing those two algorithms.

2.1 Using GPUs for General purpose computing

GPUs were originally invented to meet the demands of 3D rendering applications. During a time when CPUs were used for graphic rendering, applications could speed up rendering operations by offloading most of the computation directly to the GPU. Impressive results delivered by GPUs led many researchers to tap into their computational power.

Availability of massive number of simple micro processors makes GPU a good candidate for parallelizing applications. In general, applications can be parallelized on GPUs with two main approaches. Either all cores in GPU can execute the same instruction or each core can execute a different instruction. For graphic rendering applications it is the first type of processing that happens. For general computing applications both the types are utilized. For computational tasks, like Matrix multiplication, each core computes a different element. But in many Data crunching applications, all the cores execute the same instruction.

Even though many applications can reap the benefits of massive computing power, not all applications running on multi-core processors can be ported onto GPUs. Che et al. [7], in their study, discuss about different domains where

GPUs can be used. They evaluate the effectiveness of CUDA as a general tool to express parallel computing on GPUs. They select five representative problems, implement on a GPU and on a high end multi-core CPU platform, and compare the performance. In many of these comparisons, running time is measured against the input size. In some of the comparisons, running time is measured by varying number of threads and cores. For comparing Data Mining applications, they mainly use number of instances as the input size.

We also take this space to review approaches followed while comparing Data Mining Applications implemented on GPUs. Similar to [7], in many GPU related studies running time has been measured against the input size. However, the studies differ by the attribute they select as the input. In [8], authors have considered both the number of instances and the length of itemsets as inputs. Experiments have been carried out by varying these two attributes. Additionally, they have also measured running time against the block size. Another study where a comprehensive list of experiments are performed is GMiner [9]. In addition to the above two inputs, they have measured running time against the number of distinct itemsets, minimum support and width of the transaction blocks. While measuring the performance, we also measured execution time against the number of instances. Since what our framework does at a basic level is counting patterns, we also measured the running time by varying the number of patterns.

2.2 Parallel Data Mining Algorithms that uses Bitmaps

In this section we mainly review work done on parallelizing Data Mining algorithms. Since there are limited studies on using GPUs for parallelizing, we have also considered studies done with CPU based many core platforms.

2.2.1 Two FIM implementations with Bitmaps

GPUs have been used abundantly for FIM algorithms, mainly because Itemset generation and support counting is inherently parallel and can be naturally aligned with GPU processing. In [10] authors present two efficient GPU based

Apriori algorithms, Pure Bitmap-based (PBI) algorithm, a one which entirely uses Bitmaps, and a Trie Based Implementation (TBI) another one that uses a Trie. PBI uses Bitmaps for both candidate generation and support counting and runs on the GPU. TBI uses a Trie for candidate generation but uses a Bitmap based structure to perform support counting. In TBI, candidate generation runs on CPU, and support counting runs on the GPU. They bring out the argument that, Trie traversal is an irregular operation which is mostly suited to a CPU. The method used by them to execute TBI, is similar to the model we are following in our framework. The data structure they manipulate, is common for both the implementations. However, they use a lookup table while obtaining 1 counts, same for which we use an instruction available on the GPU. Authors also highlight the fact that when number of items are huge, the data structure they are using tend to make non-coalesced memory accesses. We have avoided this in our framework by selecting the direction with most elements as the vertical direction.

Another couple of studies where GPUs are used in FIM algorithms are [8] and [9].

2.2.2 gpuDCI

In [8], authors propose a FIM implementation which defers using GPU until all the frequent itemsets fit into device's memory. This technique prevents frequent data transfers between host and the device. In this study, the authors explore two parallelizing strategies, Transaction-wise parallelization and Candidate-wise parallelization. In the former, all cores of the GPU process elements belonging to the same two Itemsets, while in the latter each individual Itemset is handled by a different core. Authors also propose a batching technique which caches results of intermediate intersections when provided a batch of Itemsets. With these techniques, they have been able to obtain a maximum speed up of 6. They further show that the techniques they propose help them to achieve better scaling. Technique we use in processing Bitmaps is the same Transaction-wise parallelizing technique proposed in this study. Authors claim that a better speed up can be

obtained when using Candidate-wise processing. But in the scope of our study since number of intersections are far lower compared to the number of data points, Transaction-wise parallelization is the suitable approach.

2.2.3 GMiner

GMiner [9] proposes a solution which partitions the Bitmaps (which represents Itemsets) into fixed length elements, which enables them process Datasets larger than the GPU memory. In this study authors have explored the possibility of using multiple GPUs while exploiting the ability to compute a partial sum in each thread block. While implementing [9], authors have studied different implementations existed at that time and have addressed scaling and load distribution issues. They highlight the fact that, the vertical Bitmap organisation is better suited for a GPU than a horizontal organisation, since the vertical organisation allows utilising GPU memory in an efficient manner. With their techniques they obtain speed ups greater than 1000 when compared with CPU algorithms.

2.3 GPU Frameworks for Data Mining Applications

2.3.1 GPUMiner

GPUMiner [6] is a Data Mining framework based on Bitmaps. They use both Horizontal and Columnar Data layouts and facilitates multiple algorithms falling into different categories. In GPUMiner, Bitmaps are being used for different types of processing. For Apriori, Bitmaps are utilised to represent unique itemsets and are used while computing support and generating candidates. From Apriori's point of view, Bitmaps are used for a core operation. But for clustering algorithms, Bitmaps are used for tracking identity of different data points, which is a secondary task compared to the distance calculating operation. GPUMiner consists of three main components, the Storage and Buffer Management component, Data Mining Component and Visualising component. Storage Components is backed by a Berkley DB, which stores Data in Chunks. Bitmaps are created on the fly as a part of algorithm execution. One important design decision they make

in implementing Algorithms is using GPUs for simple and regular processing and doing irregular type of processing with CPU. This does make data transfers more frequent between CPU and GPU, and their method to make those transfers small is to only transfer the result back to CPU while keeping the Bitmap structure on the GPU. This is the same design we used in our framework.

Authors compare their Apriori implementation with FIMI03 [11] and K-means implementation with an algorithm known as UVirginia. GPUMiner reports a maximum speed up of 10.4 against FIMI03 and a speed up of 5 against UVirginia. While testing Apriori, they compare against a CPU based Bitmap variant, in addition to FIMI03. This Bitmap variant reports a better speed up than FIMI03. Since the GPU variant also uses Bitmaps, it is not clear which element is responsible for the speed up; whether it is the parallel processing or it is the Bitmap organisation.

2.3.2 Index Structure for Similarity Joins

Böhm et al. [12], proposes a framework which uses a multilevel index Structure that speeds up similarity Joins. Similarity Join evaluates the degree to which one data point is similar to another and presents a set of points with a difference less than a given threshold. The algorithm starts by building the index structure in a bottom wise fashion, which would sort the Data set and create high level partition by a single attribute. Partitions will be further sorted by other attributes and sub partitions will be created. They demonstrate how this index structure can be used to speed up similarity join. The nested loop join, which is the version of Similarity Join performed without the index structure, would iterate through entire Data set in a point by point fashion. Evaluation happens in two iterations; one outer loop picking the first point to be compared and another inner loop selecting the second point and evaluating the threshold. With the index structure, they limit the iterations in the inner loop because now they can find collocated points by loading a partition.

They perform several experiments with their index structure. They compare

the efficiency of their index structure by comparing it with Nested Loop Join on both CPU and GPU. And then they implement two clustering algorithms DBSCAN and K-Means by expressing those with Similarity Join. They claim that speed ups around 100 are achievable with their index structure. Even though they propose this as a generic index which can be used with ARM and Classification Algorithms, so far it has only been used with Clustering Algorithms.

Similarity between their approach and ours is that in both the studies algorithms are implemented around a core operation, and by speeding up the core operation, execution time is improved. In our framework it is by using a technique popular in FIM algorithms we speed up two Classification algorithms. So in terms of extensibility, since the methods we use are widely used in FIM algorithms we have more evidence to claim that our framework can be used across Associate Rule Mining and Classification algorithms.

2.3.3 Framework for Mapping Data Mining Applications on GPUs

Another study that proposes a framework for Data Mining algorithms is [5], where the framework optimises large data transfers required by algorithms. Authors observe that by grouping Data points which gets processed together and by pre-fetching data points, overall execution time can be reduced. The framework consists of three main components, a Storage Component which handles memory transfers from Disk to Host's Main memory and then to Device Memory, a Scheduling component which schedules Tasks by minimising data dependencies by each thread and finally the Mining component where all algorithms specific optimisations can be included. To evaluate their framework, they compare it with GPUMiner and with a custom implementation they did using Bitmaps. On some data sets their framework shows a 3 percent improvement against GPUMiner, but in some cases GPUMiner shows better results than the framework. In most of the cases however, their custom Bitmap implementation shows the best improvement which reports a maximum of a 14 percent gain. With their approach of unifying data transfers, they could use optimization techniques applied for one algorithm

to improve another, but they have not been able to surpass speed ups that can be obtained with Bitmaps.

2.3.4 Three techniques to improve Data Mining algorithms

In work done by Jian et al. [13] they propose 3 main techniques which improves processing on GPUs. These techniques address three main problems occurring in Data Mining applications. Their solution to process high-dimensional data is to follow column-wise processing which enables GPU to apply sequential addressing reduction [14], which is the same technique we use in our study.

2.4 GPU Implementations for Naïve Bayes and Decision Trees

Since we implement Naïve Bayes and Decision Trees using Bitmaps, we reviewed previous studies done on the same.

Recently, Viegas et al. [15] implemented Naïve Bayes algorithm on GPUs. In their implementation they used a compact data structure indexed by terms, which helped them to minimize memory consumption. In their implementation, they perform both model building and classification on GPUs. The compact structure being used, help them to perform model building in parallel, allowing them to achieve 35 times speed up over sequential CPU execution. The index structure proposed is specially crafted for Automatic Document Classification and they do not claim their implementation as a generic Naïve Bayes implementation. In our approach we present a generic, Naïve Bayes implementation which only uses a Bitmap-based structure to keep the underlying Dataset.

One of the earliest methods for building a Decision tree in parallel has been proposed in SPRINT [16], where records are distributed among multiple processors. Algorithm SPRINT is an improvement over SLIQ [17], and has adopted many characteristics from SLIQ. SPRINT proposes a parallel tree building technique which parallelize computation by delegating each node to a different processor. But these algorithms are designed on multiprocessor systems, where each processor has access to a dedicated memory and a hard disks. At a conceptual

level, data organisation and processing followed in our framework for Decision Tree, is similar to the methods used in SPRINT [16]. In SPRINT, each multiprocessor receives a data partition with equal number of records. During the execution each multiprocessor produces a local count matrix which is used to determine the split point. In our framework, each GPU core processes a segment of the original data column. Authors claim that this arrangement makes SPRINT scale better with increasing Data volumes. Since our framework allocates a balanced workload across multiple GPU cores, we show that the framework scales better as Data volume increases. Authors compare SPRINT with two variants of SLIQ. Against SLIQ/D and SLIQ/R they get speed ups of 16 and 4 respectively.

Techniques proposed in [16] have been adopted in CudaTree [18] which is a GPU based implementation. In addition to the characteristics borrowed from SPRINT, another approach CudaTree [18] explores is, blending task parallelism and data parallelism by switching between two different modes of tree building. They further state that data parallelism alone, increases computing overhead when processing leaf nodes. We experienced the same problem, but could not find a means to mitigate the problem.

We could not find a study which used GPUs to implement a Decision Tree. The above two implementations are those that came closest to Decision Trees implemented on GPUs. To the best of our knowledge, this is the first study that uses a Bitmap based implementation on GPUs to speed up Decision Trees.

In this chapter, we reviewed previous work done on Data Mining Algorithms implemented using Bitmaps and GPUs. Since using Bitmaps is a main component of our study, we gave prominence to Bitmap based frameworks. One of the important design decision proposed in previous studies is minimizing data transfers between CPU and GPU. And in couple of studies authors have highlighted the importance of using GPUs for regular processing and leaving more complex processing to CPU. We have incorporate these techniques while implementing our framework.

Chapter 3

METHODOLOGY

This section mainly describes the design and implementation of our framework. First we give a high level overview about the framework, elaborating different layers and core components. Then we explain about the Bitmap representations we use in the framework which are used in Data Storage layer. We move onto describe about core processing algorithms, which runs on the Bitmap Data Structure and perform some sort of a counting operation. These algorithms provide a basis for higher level algorithms. We discuss in-depth about the two Bitmap variants supported by the framework, Bitmaps and Bit-Slices, highlighting each area they can be optimally used in. We also talk about the two algorithms implemented using the framework, Naïve Bayes and Decision tree, detailing about types of processing needed for each algorithm and showing how our framework provides those. Then we move onto discuss how batching is implemented and how it reduces running time of algorithms.

3.1 Architecture

Framework can be logically separated into three main layers, Data Management Layer, Core Processing Layer and Algorithm Layer. Data Management Layer is backed by a Data Structure which is based on Bit-Slices (a Bitmap variant). Framework is capable of switching between both Bitmaps and Bit-Slices, but by default, Data will be kept in Bit-Slices, since Bit-Slices are capable of representing both numerical and categorical attribute values. Data Management layer is responsible for converting a raw Data set to the underlying representation. This layer also provides methods to transfer data between host and the device.

The next layer, Core Processing Layer is the one that directly interacts with the Data Structure and give the result for a query. When data is represented with Bitmaps, applying filters and searching for data elements needs Bitmap ma-

nipulation. We will cover some details in a latter section on how different this processing is from normal element wise processing. Due to these differences, obtaining a count with a Bitmap structure is not straightforward. This is why we are providing another Layer which performs the Bitmap processing and presents a simpler interface to Algorithm implementer. This layer provides a set of Core Algorithms, which would run on the GPU and perform certain high level operations. ElementCount, Co-occuranceCount, RangeCount are some of the algorithms provided in this layer. ElementCount would return the count of a given element. Co-occuranceCount would count the co-occurrence of two numbers in two columns. And RangeCount would perform a range query and return the number of records satisfying the condition.

The final Layer is the Data Mining layer, where high level Algorithms are implemented. These algorithms would use counting core operations implemented in previous layer to query the Data set and obtain results needed to build the model. In our approach the main separation is between the part that queries data and the part that builds the model. Usually, a Data Mining algorithm would have to perform different types of processing like recursions, tree traversals while building a model. And sometimes, frequent memory accesses are needed while sharing states across different stages. For these types of processing CPUs would suit more, since instruction pipelines and caches in CPUs are better tailored to handle these scenarios. We are using GPUs only for Bitmap manipulation which is a SIMD operation, for which GPUs are good at.

At the beginning of an algorithm execution, the entire Data set gets copied onto the Device memory. Then the algorithm would run on the CPU and at points where counts are needed, kernel functions get invoked. Since GPU is only transferring result of a computation, there would not be any major data transfers from GPU to CPU. For all algorithms, Bitmap intersection and counting will be performed by GPU, while CPU maintains the counts and perform other execution paths.

3.2 Bit-Slice And Bitmap Representations

Framework uses two representations to encode data which are explained in detail in the following sections. Before converting to either format, data is first arranged into a column-major format.

The representations we refer by the terms Bit-Slices and Bitmaps are widely known index schemes available in literature. For the purpose of our work, rather than using as index schemes we are using those to store underlying data.

The bitmap representation is similar to Value-List indices proposed in [19]. If Data set D can be represented as a collection of Attributes $\{A_1, A_2, A_3, \ldots, A_{\mathrm{n}}\}$ where each Attribute has $\mid R \mid$ number of elements and attribute cardinalities or the number of distinct values in each attribute can be expressed as $\{C_1, C_2, C_3, \ldots, C_{\mathrm{n}}\}$, then we can define bitmap and bit slice representations as below.

The Bitmap representation is a Set B $\{B_1, B_2, B_3, \ldots, B_{\mathrm{n}}\}$ where B_{i} is the set of Bitmaps corresponding to Attribute A_{i}. B_{i} can be expressed by a set of Bitmaps $\{b_{\mathrm{i,1}}, b_{\mathrm{i,2}}, b_{\mathrm{i,3}}, \ldots, b_{\mathrm{i,m}}\}$ where $b_{\mathrm{i,j}}$ is a vector of bits consisting of either ones or zeros and $m = C_{\mathrm{i}}$. Size of each Bitmap is equal to the number of records in the Data set or $\mid b_{\mathrm{i,j}} \mid = \mid R \mid$. Assuming that distinct values in A_{i} can be expressed by the set $\{a_{\mathrm{i,1}}, a_{\mathrm{i,2}}, a_{\mathrm{i,3}}, \ldots, a_{\mathrm{i,m}}\}$, then k^{th} bit in $b_{\mathrm{i,j}}$ is set to one only if k^{th} value in A_1 is equal to $a_{\mathrm{i,j}}$. This way k^{th} value will be set to 1 only in one bitmap. Loosely defining, $b_{\mathrm{i,j}}$ gives the locations $a_{\mathrm{i,j}}$ is appearing in Data set. Fig. 3.1 gives a graphical illustration of the Bitmap representation.

Bit-Slice representation of Data set D can be defined by, making slight modification to the previous. Assuming attribute A_{i} can be represented as a binary number with $N + l$ bits, the Bit-Slice representation of A_{i} is an ordered list of bitmaps $b_{\mathrm{i,N}}$, $b_{\mathrm{i,N\text{-}1}}, \ldots, b_{\mathrm{i,1}}, b_{\mathrm{i,0}}$ where these bitmaps are called the Bit-slices. If $A_{\mathrm{i}}[k]$ denotes, k^{th} element in Attribute A_{i} and the bit for row k in Bit-Slice $b_{\mathrm{i,j}}$

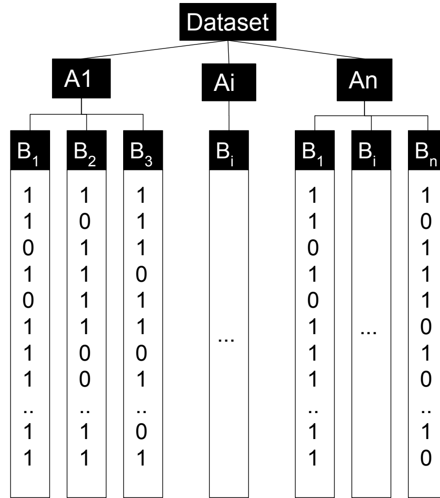

Figure 3.1: Data set represented using Bitmaps.

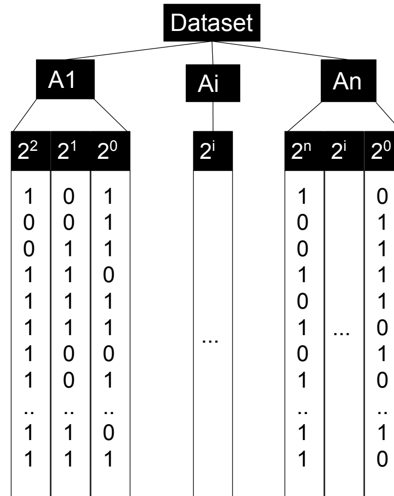

Figure 3.2: Data set represented using Bit-Slices.

by $b_{\mathrm{i,j}}[k]$ then the values for $b_{\mathrm{i,j}}[k]$ are chosen so that

$$A_{\mathrm{i}}[k] = \sum_{i=1}^{N} b_{\mathrm{i,j}}[k] \times 2^{i} \tag{3.1}$$

Note that we determine N in advance so that the highest-order Bit-Slice $b_{\mathrm{i,N}}$ is non-empty. Usually N is selected so that $N = log_2(max(A_{\mathrm{i}}))$. Bit-Slice representation of the Data set D is the set $B\ \{B_1, B_2, B_3, \ldots, B_{\mathrm{n}}\}$ where B_{i} is the Bit-Slice representation of Attribute A_{i}. Fig. 3.2 illustrates the Data sets represented as Bit-Slices. Note that all the values in column A_1 is represented by the three Bit-Slices.

In the framework, by default Data is represented in Bit-Slices, for its ability to represent data with a smaller volume. Each representation can be switched to the other without referring to the original Data set.

Even we define a single Bitmap as a vector of bits, when implementing it, bits are grouped into chunks of 64 and is usually stored as an array of *ulong* or *int*s. Then Bitmap intersection would reduce into performing bitwise AND between two *ulong* arrays.

In this thesis we might use the term Bitmap in different contexts. First we use it to refer to the Bitmap representation, which can be used for Categorical Data sets. We also use it to refer to a single array of ulong literals that makes up an individual entity (or a Bitmap). When we talk about existential Bitmap it is in the latter sense we use the term. The Bit-Slice representation consists of a collection of Bit-Slices. And sometimes we use the term Bitmap instead of Bit-Slice since its more natural to use.

3.3 Processing Bit-Slices on a CPU

To understand how we can perform search and count operations with Bit-Slices we are first going to show how it is done on a CPU environment. Once we have an idea how processing is done, it will be easier to visualise how it happens on the GPU.

3.3.1 Counting a single element with Bit-Slices

Doing operations with Bitmap representation is straightforward since Bitmaps are created for each category and obtaining count with Bitmap only needs intersecting with another Bitmap. But processing Bit-Slices is not as simple, since it encodes a range of numbers. Operations needed to select a single number from a column encoded in Bit-Slices is shown in Algorithm 4.2 [19]. But to illustrate this clearly we are presenting the following simple example. Assume that we have a Data set consisted of a single column, ItemNo as depicted in figure 3.3. By arranging all the elements of the column into a single array we form the columnar (or column

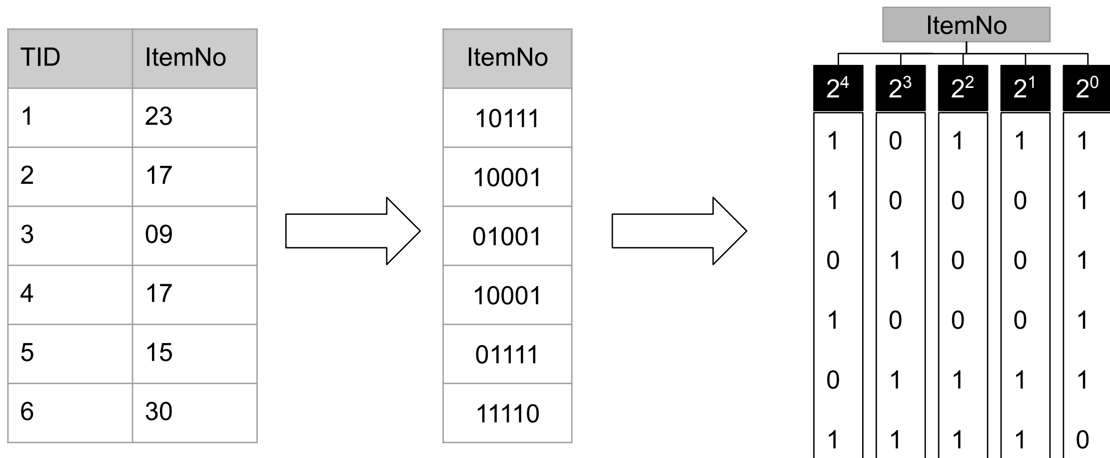

Figure 3.3: Converting a single column to Bit-Slices.

wise) representation of the same Data set. Now to convert this into it's Bit-Slice representation we first get the binary representation of all the elements. Then we take all the bits falling under a particular position (let's say 0th position) and call it a single Bit-Slice.

The next figure 3.4 shows how an element is actually looked up in the Data set. Let's say that we need to select number 17 from the column. We would not be able to iterate element by element because now the data is represented in a different format. Instead, what we do is we formulate a series of operations to be performed on Bit-Slices such that once they are applied, the resulting Bitmap only have the rows with desired number set to 1. For this example, the selected number is 17 which has its binary representation as 10001. The sequence of operation that would only select this particular number is $1 \,\&\, !0 \,\&\, !0 \,\&\, !0 \,\&\, 1$. Once this sequence is applied we get a resulting Bit-Slice, which have particular locations where 17 was in, set to 1. By counting 1s in this Bitmap we get the count of 17s.

3.3.2 Counting co-occurrence of two elements

Co-occurrence counting is a frequently used operation since this is used when populating contingency tables. While implementing Both Naïve Bayes and Decision Tree we used a contingency table to determine probabilities and InfoGains.

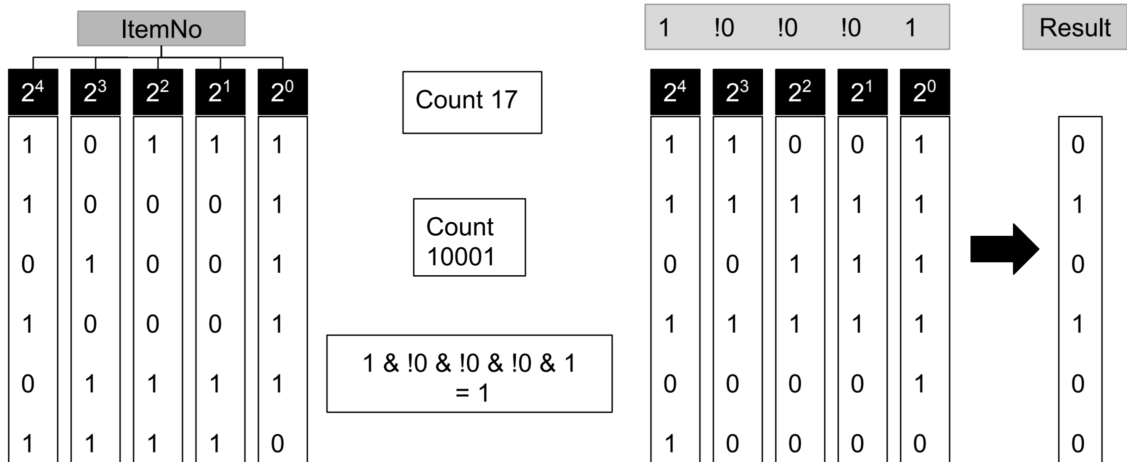

Figure 3.4: Counting occurrences of a number with a Bit-Slice column.

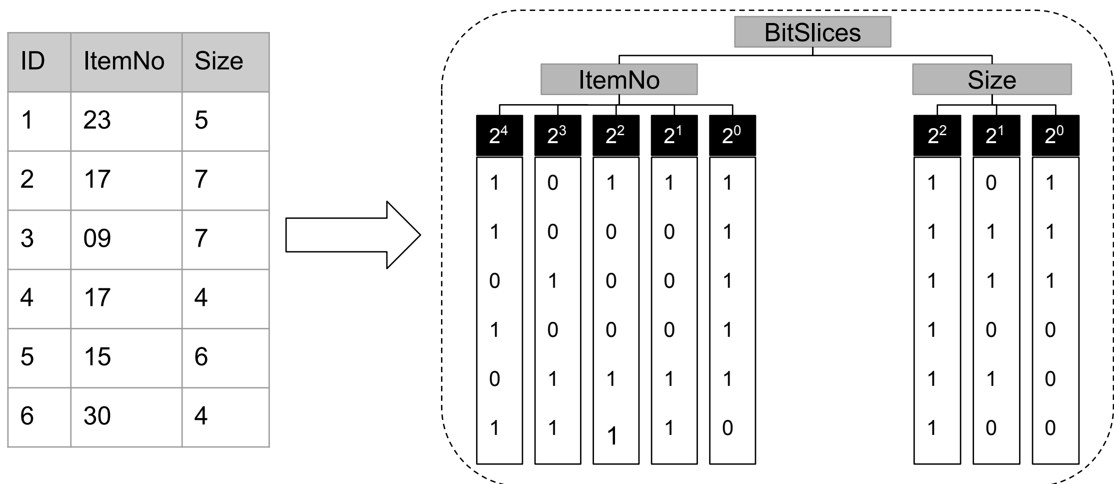

Figure 3.5: Converting two columns to Bit-Slice representation.

To illustrate this, we can use the same example we used earlier with a minor modification. We are going to add another column Size to the same Data set. With this we can generate two Bit-Slice columns as shown in 3.5.

We are going to find the count of rows satisfying the condition ItemNo=17 & Size=7. As depicted in the Figure 3.6 we are going to generate a Bitmap from each Bit-Slice column by doing a single element match and then intersect the two resulting Bitmaps to get the final result.

The use of two Bitmaps was shown to help with visualising operation easily. Algorithm 1, shows how this is actually done.

Here we invoke the procedure Co-OccurrenceCount, passing Data set repre-

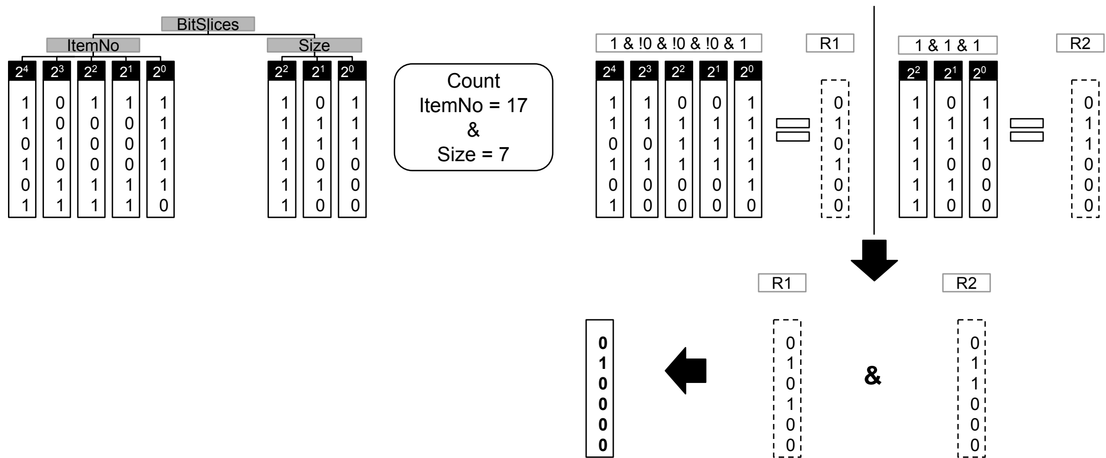

Figure 3.6: Counting co-occurrences with Bit-Slices.

sented in Bit-Slices as an argument. We also pass $val1$ which is the value that needs to be searched from the first column, $index1$ which indicates the index of the first column. $val2$ and $index2$ bears similar meanings for the second column. In lines 2-3 we assign the columns we are going to process to two variables $col1$ and $col2$. Contents of these variables can be thought of as a two dimensional vector where the first dimension selects the Bit-Slice by the index and the second dimension selects the chunk of bits (the $ulong$ value which groups a chunk of 64). Then we initialise a temporary array (of $ulong$s) which would hold the result of the intersection. Length of this would be equal to the length of any Bit-Slice. We also initialise two other variables to hold number of bits of each value we are searching, which will be used while iterating through Bit-Slices.

In the block from 8-14, the first Bit-Slice column is looked up. We use left shift operator in line 9, to determine whether a particular bit is set or not. If k^{th} bit is 1 for $val1$ then $val1 \,\&\, (1 << k)$ would evaluate to true. It is based on this values we determine whether to do a simple intersection or an intersection with a negation. The inner loop from 10-11, corresponds to iterating through a Bit-Slice while performing the intersection and writing the result to the temporary array. In a similar way we do the lookup for $val2$ in second column. Note that without initialising a second variable to store the result, we are using the same temp variable defined in line 4.

Algorithm 1 Algorithm for Counting Co-Occurrences

```
 1: procedure Co-OccurrenceCount(BitSlices, val1, index1, val2, index2)
 2:     col1 ← BitSlices[index1]
 3:     col2 ← BitSlices[index2]
 4:     temp ← ulong[length(col1[0])]
 5:     bitlength1 ← log2(val1)
 6:     bitlength2 ← log2(val2)
 7:     sum ← 0
 8:     for k ← 0 to bitlength1 − 1 do                ▷ Matching first Element
 9:         if val1 & (1 << k) then
10:             for i ← 0 to length(col1[0]) − 1 do
11:                 temp[i] ← temp[i] & col1[k][i]
12:         else
13:             for i ← 0 to length(col1[0]) − 1 do
14:                 temp[i] ← temp[i] & !col1[k][i]
15:     for k ← 0 to bitlength2 − 1 do                ▷ Matching second Element
16:         if val2 & (1 << k) then
17:             for i ← 0 to length(col2[0]) − 1 do
18:                 temp[i] ← temp[i] & col2[k][i]
19:         else
20:             for i ← 0 to length(col2[0]) − 1 do
21:                 temp[i] ← temp[i] & !col2[k][i]
22:     for k ← 0 to length(temp) − 1 do              ▷ Counting and summing up
23:         sum ← sum + popcount(temp[k])
        return sum
```

By the beginning of line 22, we have gone through both the columns and have produced a single resulting Bitmap containing result of both the intersections. From line 22-23 we iterate through this resulting Bitmap to do the count of 1s. We are using the population count instruction (*popcount*) which returns the no of 1s in a particular *ulong* , which is supported in most CPUs and GPUs. Other option is to use a lookup table which is much slow compared to the approach we are using.

3.4 Processing Bitmaps And Bit-Slices on GPU

Since we already know how processing happens in CPU, this section will explain how the processing happens on the GPU.

The basic unit in either of these representations is a Bitmap, which is kept as *ulong* vector. Result of an intersection produces another Bitmap, count of 1 of which can be obtained using *popcount* instruction available in CUDA. Since both Bitmaps and BitSlices are similar in representations, we will explain in detail about Bitmap processing and then briefly discuss about Bit-Slices.

3.4.1 Processing Bitmaps on GPU

With the Bitmap representation each attribute is a collection of Bitmaps, so counting co-occurrence between unique values in two attributes would involve intersecting and counting Bitmaps.

In sequential addressing reduction, each GPU core would work on the same intersection and count operation, regardless of the GPU multiprocessor they belong to. Each thread is in charge of an interleaved portion of the Bitmap, in such a way that threads having consecutive indexes work on consecutive parts of the Bitmap. Since these Bitmaps are vectors of *ulong*s, each thread processes a contiguous chunk of 64 bits, separated by a fixed stride. Usually this stride is taken as the multiplication number of thread blocks and threads per blocks, which are decided at kernel launch time. Each thread would read two Bitmaps do the intersection and then the count operation (*popcount*) and write result to

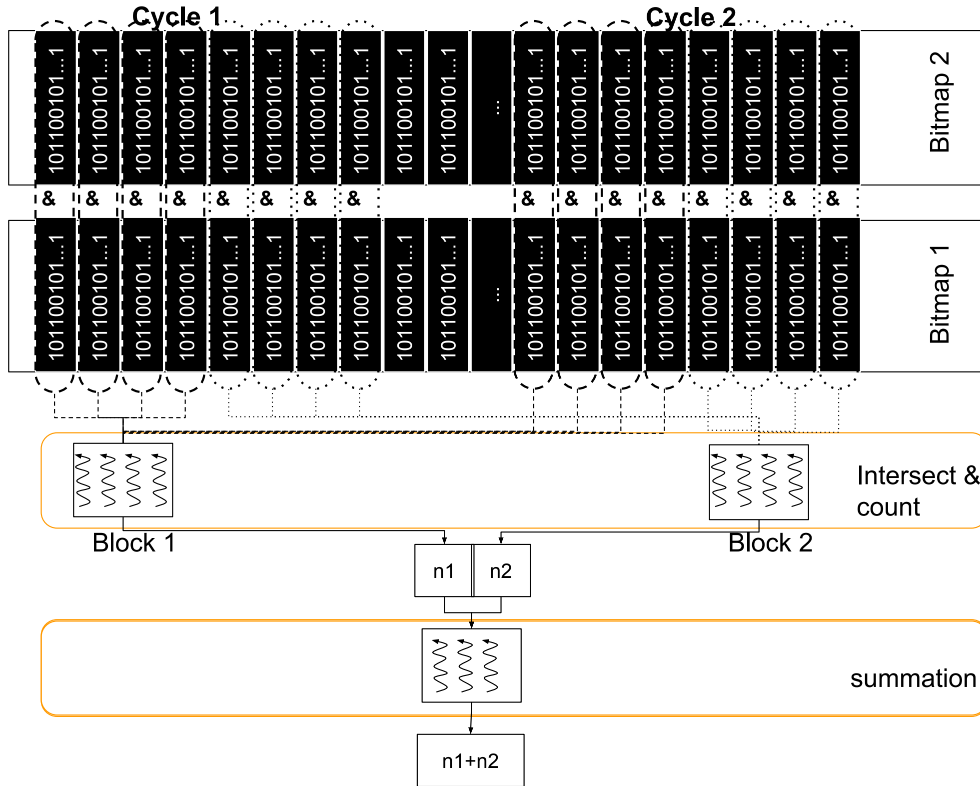

Figure 3.7: Bitmap intersection & counting on GPU.

its shared memory, through which we get a local aggregation of the counts by the block. At the end of its execution, each block would write results for local aggregation to a global array, which can be either reduced by copying to the Host or invoking a second kernel which only performs a simple reduction. The kernel would only write to the global memory at the time of finishing kernel invocation, and at other times it would only do reads. When doing reads, consecutive threads will be read from adjacent memory locations so the accesses are coalesced. And when writing to shared memory, a sequential addressing method is followed to avoid bank conflicts.

In Fig. 3.7 we provide a visualisation of how Bitmap intersection and counting happens on the GPU. Bitmap 1 and Bitmap 2 are two *ulong* arrays residing in main memory of the GPU. In *Cycle* 1 each block will be processing the set of elements located to the left of the Bitmaps. *Block* 1 will be processing elements demarcated by broken lines while *Block* 2 will be processing elements with dotted lines. Each thread in the block will pick an index and read two elements located

at that position from two Bitmaps. Intersection and counting would happen in each thread and would get aggregated by the block level when writing to shared memory. At the end of $Cycle\,1$, $Block\,1$ will write the aggregation of 4 elements located to the very left of the Bitmap and Block 2 will similarly write down aggregation of the next 4. In $Cycle\,2$, both the blocks will pick a different portion of the same Bitmaps. $Block\,2$ will be picking the last 4 elements to the right, the ones marked with dotted lines and $Block\,1$ will pick next 4 elements from the end marked with broken lines. The value $n1$ provided by $Block\,1$ at the end of $Cycle\,2$ is the aggregation of all elements processed by $Block\,1$. Similarly $n2$ is the aggregation of all elements processed by $Block\,2$. If there are n blocks, then an array of n will be written to Global memory, each with the aggregation of all elements processed by each block. Summing up this array would give the result for the entire Bitmap. This is usually done by running a summing kernel providing the array with partial sums as the input.

The Algorithm 2 shows how the kernel performs Bitmap intersection. Here *col*1 and *col*2 are the respective columns (attributes) represented in Bitmaps needed for the intersection. Each column can be thought of as an array of Bitmaps. Since we are only representing categorical data with Bitmaps, a single category value would have a unique Bitmap. $index1$ and $index2$ are the indices of the first and second category values respectively. We also pass the $length$ which gives the number of $ulong$ literals in a Bitmap. We first initialise internal state variables by getting Block and Thread configurations. The variables *threadIdx* and *blockIdx* are set by CUDA environment based on parameters we set while invoking the kernel. Since this kernel is invoked by each thread, each thread needs to select a non-overlapping portion of the Bitmap. That is why the variable i is determined using $blockIdx$ and $threadIdx$. With Bitmaps processing is simple. Once we initialize variables properly, we do a Bitmap intersection in line 9. A single thread may work on multiple portions in different iterations. To enable this we keep increasing i by the size of the Grid (i.e the number of blocks). We have omitted certain optimisations for the sake of clarity. Even though we are only showing one intersection here, in actual kernel, there are two. Also the index

resolution in Bitmap array happens in a different way. We have also omitted the part which performs block-wise aggregation.

Algorithm 2 BitmapCo-OccurrenceCountGPU

```
1: procedure BitmapCo-OccurrenceCountGPU(col1, col2, length, index1,
   index2,output)
2:     tid ← threadIdx.x
3:     i ← blockIdx.x × blockSize + threadIdx.x
4:     gridSize ← blockSize × gridDim.x
5:     sdata ← initialize shared memory
6:     mySum ← 0
7:     bitwise ← 0
8:     while i < length do
9:         bitwise ← col1[index1][i] & col2[index2][i]
10:        mySum ← mySum + __popcll(bitwise)
11:        i ← i + gridSize
12:    sdata[tid] ← mySum
13:    ...                                    ▷ Omitted reduction steps for clarity
14:    if tid == 0 then        ▷ Only one thread will write the final value to Global
Memory
15:        output[blockIdx.x] ← mySum
```

3.4.2 Bit-Slice Processing on GPU

Main difference between Bitmap and Bit-Slice manipulation is that in Bit-Slices the entire set of Bitmaps belonging to the two attributes need to be loaded as opposed to reading the two particular Bitmaps.

Algorithm 3 show the Bit-Slice intersecting kernel running on GPU. Apart from couple of places this is very much similar to the CPU algorithm. This is different from CPU algorithm since there we pass the entire Data set and the two column indices, as opposed to passing the exact columns that needs to be processed. Additionally, we pass the number of Bitmaps in each column (indicated by *col1_bitmaps* & *col2_bitmaps*) and a pointer to output array residing in the Global Memory. Similar to 2, *col*1 and *col*2 are columns having all the Bit-Slices belonging to that column. Bit-Slice Intersection looks a bit complex compared to Bitmap Kernel, since there is loop running to select the number. In Bitmap Kernel we did not need to explicitly pass the category values, but for Bit-Slices,

Algorithm 3 Bit-SliceCo-OccurrenceCountGPU

```
1: procedure Bit-SliceCo-OccurrenceCountGPU(col1, col2, length, val1, val2
   , col1_bitmaps,col2_bitmaps,output)
2:     tid ← threadIdx.x
3:     i ← blockIdx.x × blockSize + threadIdx.x
4:     gridSize ← blockSize × gridDim.x
5:     sdata ← initialize shared memory
6:     mySum ← 0
7:     bitwise ← 0
8:     while i < length do
9:         bitwise ← 0
10:        for k ← 0 to col1_bitmaps − 1 do
11:            if val1 & (1 « k) > then
12:                bitwise ← bitwise & col1[k][i]
13:            else
14:                bitwise ← bitwise & !col1[k][i]
15:        for k ← 0 to col2_bitmaps − 1 do
16:            if val2 & (1 « k) > then
17:                bitwise ← bitwise & col2[k][i]
18:            else
19:                bitwise ← bitwise & !col2[k][i]
20:        i ← i + gridSize
21:        mySum ← mySum + __popcll(bitwise)
22:    sdata[tid] ← mySum
23:    . . .
24:    if tid == 0 then
25:        output[blockIdx.x] ← mySum
```

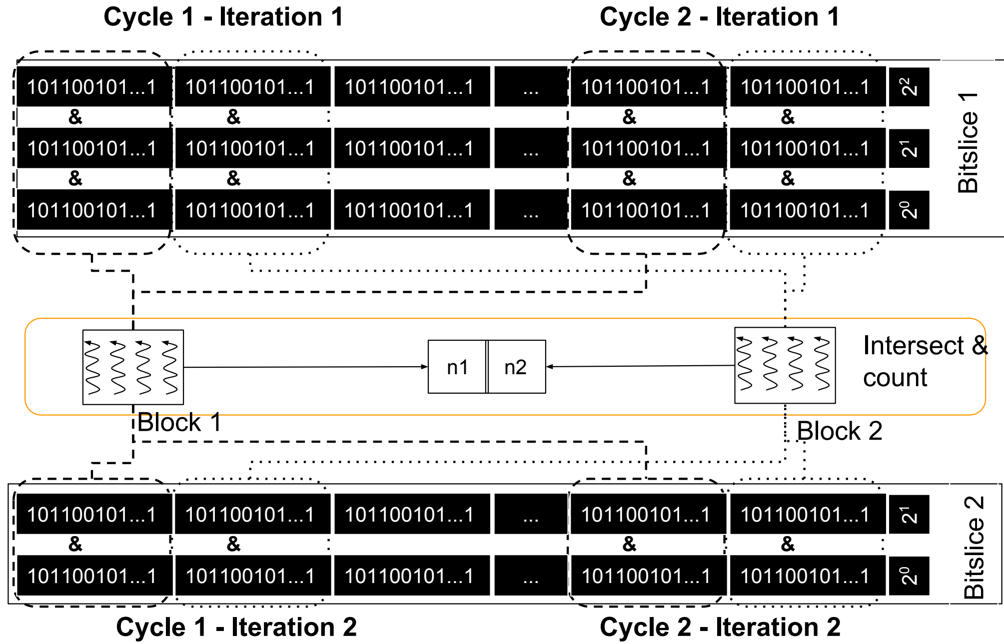

Figure 3.8: Bit-Slice intersection & counting on GPU.

we need since it is based on those values intersection is done. This is graphically illustrated in 3.8.

Similar to Bitmaps, once the intersecting and counting is done, a block level reduction happens, which is followed by a global reduction. The only noticeable difference between the two representations is that, in Bitmap representation a Bitmap is readily available for an attribute value, but in Bit-Slices it will be generated by the kernel as computation happens.

3.4.3 Batching operations

Usually arithmetic intensity of a Bitmap intersection is low. To produce intersection of two Bitmaps, two global reads have to be made. This makes the kernel, a bandwidth sensitive one since a larger proportion of time is spent in transferring Bitmaps from global memory. With batching we will be reading four Bitmaps to produce four resulting Bitmaps. So four co-occurrences can be counted with this approach.

Recall that in Bitmap intersection once the 1-count is done, it is aggregated in block level which is followed by a global reduction. In non-batched execution, 1-count is held by an integer. However with a single integer we can only hold the result of a single co-occurrence. So to accommodate multiple patterns we use *int4* literals. Local and global reductions would happen the same way it happened for a normal int array. However before transferring results to host's side they need to be converted into proper integers.

Even we are only referring to co-occurrence count, batching is not limited to that operation. It is equally applicable to the other two operations.

3.5 Algorithm Execution

This section explains how the two algorithms were implemented on our framework. We first go through the types of processing needed by each algorithm. Then we show how we can convert those operations to counting operations and how those counts can be obtained using our framework.

3.5.1 Implementing Naïve Bayes on framework

Bayes theorem states that the relationship between the probability of Hypothesis before getting the evidence $P(H)$ and the probability of the hypothesis after getting the evidence $P(H \mid E)$ is :

$$P(H \mid E) = \frac{P(E \mid H)\, P(H)}{P(E)} \tag{3.2}$$

Naïve Bayes algorithm is based on the above theorem that calculates the probability of a record r_{t} to be classified as c_{i}, based on probability of c_{i} occurring in the Data set and and the likelihood of data r_{t} occurring with c_{i}. More formally, if $P(c_{\mathrm{i}} \mid r_{\mathrm{t}})$ is the probability that a record r_{t} $(a_1, a_2, \ldots, a_{\mathrm{n}})$ belongs to class c_{i} , then using Bayes Theorem probability is calculated as:

$$P(c_{\mathrm{i}} \mid r_{\mathrm{t}}) = \frac{P(r_{\mathrm{t}} \mid c_{\mathrm{i}})\, P(c_{\mathrm{i}})}{P(r_{\mathrm{t}})} \tag{3.3}$$

Since the cost of calculating $P(c_i \mid r_t)$ increases as the possible vectors r_t increase, the Naïve-Bayes algorithm simplifies calculation by assuming that the occurrence of all attribute values are independent. With this each attribute value a_j can be calculated independently from the others. Incorporating this Assumption, $P(c_i \mid r_t)$ can be re-written as:

$$P(c_i \mid r_t) = \frac{P(c_i) \prod_{j=1}^{n} P(a_j \mid c_i)}{P(r_t)} \tag{3.4}$$

Assuming a multinomial distribution, $P(a_j \mid c_i)$ can be calculated by Eq. 3.5, where $N(a_j \mid c_i)$ is the number of times, attribute a_j is co-occurring with class value c_i.

$$P(a_j \mid c_i) = \frac{N(a_j \mid c_i)}{N(c_i)} \tag{3.5}$$

Now thinking of a categorical Data set which is arranged into Bitmaps, where each category value is represented by a single Bitmap, $N(c_i)$ will be equivalent to taking count of the Bitmap that represent c_i. Similarly, $N(a_j \mid c_i)$ will be equivalent to intersecting bitmaps representing a_j and c_i and then taking the count. To find the most probable class given a data point r_t, we need to find the maximum of the multiplication between probability of r_t given c_i and the prior probability of c_i.

$$\widehat{c} = \underset{c_i \in C}{argmax}\ P(c_i)\ P(r_t \mid c_i) \tag{3.6}$$

If we have the counts for co-occurrence between each attribute value with class attribute values, and count of each attribute value separately, we can calculate the above probabilities and do the classification. Basically, if the contingency table for the Data set with class attribute can be computed, we can easily classify unseen instances.

3.5.2 Implementing Naïve Bayes

To make a fair basis for comparison, we used the implementation provided by Weka [20]. Additionally, Weka implements all Algorithms on a common data

storage, which closely aligns with our requirement. The algorithm starts by initializing a two-dimensional matrix at hosts' memory space, which is a portion of the complete contingency table. The contingency table is built in several iterations, so in a given iteration, counts associated with a selected attribute with class attribute is obtained. At the beginning of the counting phase Data set is transferred to main memory of Device, and counting kernels are invoked. The 2D Matrix has cells equal to the $att_values \times class_values$, where att_values and $class_values$ represent number of distinct values of the test attribute and class attribute respectively. In non-batched mode, each kernel invocation counts a single pattern requiring that many invocations. At the end of each kernel invocation, a single cell in the matrix gets filled. The batched mode can count four patterns at once, so depending on the Data set, a minimum of one fourth of the invocations will happen.

3.5.3 Implementing Decision Trees on framework

A Decision Tree is a classifier that partitions data recursively into groups or classes until each partition consists entirely or dominantly of instances from one class.

Usually there are two distinct phases, the tree building phase where tree is grown by repeatedly partitioning the training set and tree pruning where some branches are eliminated to avoid over fitting. During the tree building phase, different algorithms are used to select the split point, and in C4.5 usually Information Gain ratio is used.

$$Entropy(t) = -\sum_{j=1}^{k} p(j \mid t) \log p(j \mid t) \tag{3.7}$$

Entropy at node t is given by $Entropy(t)$ where $p(j \mid t)$ is the relative frequency of class j at node t and k is the number of classes. If n_{t} records are available at node t and $n_{\mathrm{j,t}}$ is the number of records having j as the class at t,

then $p(j \mid t) = \frac{n_{\mathrm{j,t}}}{n_{\mathrm{t}}}$. Similarly Information gain at node t can be defined as;

$$Gain(t) = Entropy(t) - \sum_{j=1}^{k} \frac{n_{\mathrm{j,t}}}{n_{\mathrm{t}}} \times Entropy(j) \tag{3.8}$$

Where Parent node t with n_{t} records is split into k partitions, and $Entropy(j)$is the entropy at split j. With this we can define $GainRatio$ at t as;

$$GainRatio(t)_{\mathrm{split}} = \frac{Gain(t)}{SplitInfo(t)} \tag{3.9}$$

Where;

$$SplitInfo(t) = -\sum_{i=1}^{k} \frac{n_{\mathrm{i,t}}}{n_{\mathrm{t}}} \log \frac{n_{\mathrm{i,t}}}{n_{\mathrm{t}}} \tag{3.10}$$

Assuming a categorical data set, where Attribute A has values $a_1, a_2, a_3, \ldots, a_{\mathrm{k}}$, then $n_{\mathrm{i,t}}$ is the number of records having attribute a_{i} as the attribute value in the partition. n_{t} is the number of total samples at node t.

3.5.4 Building Decision Trees with Bitmaps

We will first describe how Decision Trees can be built with Bitmaps, and then explain how it can be ported to a GPU. The version that entirely runs on the CPU is similar to what is described in [21].

For the ease of understanding it, let's assume we are building tree from the root level and we have a Data set which have $A1$, $A2$ as the attributes and C as the class attribute. Cardinality of the attributes are k,l and m. Figure 3.9

To determine the split we would have to calculate information gain. In order to calculate info gain, we need entropy at the root level, and entropy obtained after splitting by each attribute. Entropy for the node can be obtained by simply doing a 1-count on Bitmaps representing each class value. To compute the entropy after splitting by an attribute, Bitmap belonging to each class value needs to be intersected with Bitmap for each attribute value. Simply by getting the contingency table between a class attribute and a test attribute, we can compute InfoGain.

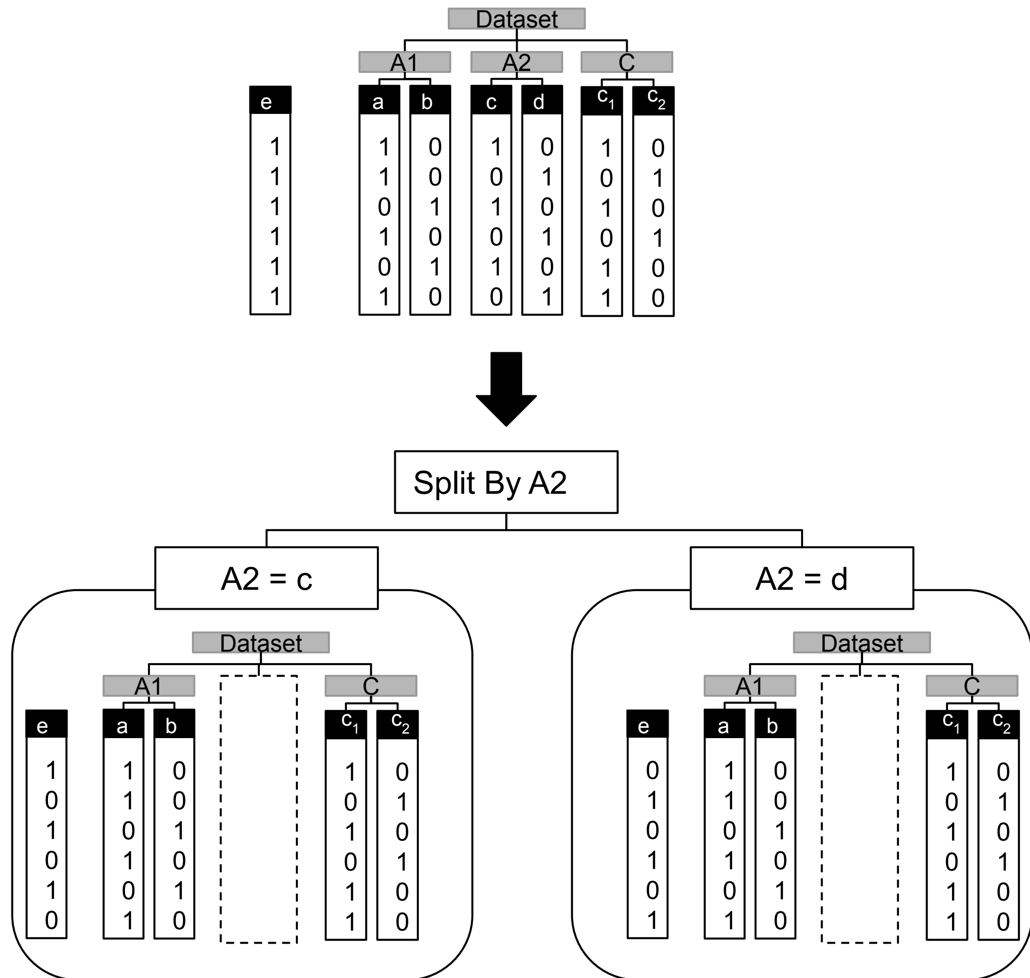

Figure 3.9: Building Decision Trees with Bitmaps.

We would additionally need SplitInfo to calculate GainRatio. And SplitInfo can be calculated by getting count of the attribute values.

Even it may appear that computing GainRatio at other partitions need more work, with the help of a simple Bitmap intersection it can be performed relatively easily. Once the the split attribute is decided, we can use the Bitmaps of the split attribute to get counts in partitions.

To better understand this, let's go through the flow illustrated in 3.9. As shown in the diagram, each attribute have two distinct values where each can be shown by a unique Bitmap. Bitmap shown by e is the existential Bitmap which keeps the presence of a row. This is needed for determining identity of an element. Now the attribute to Split is set as $A2$. Now we would have to create two splits

using the Bitmaps of $A2$. What we do is, we intersect the existential Bitmap of Data set with Bitmaps for each Attribute value and create a set of partition-wise Bitmaps. When calculating Entropy for the partition by c (left partition), resulting Bitmaps (the ones generated by intersecting $A1$'s Bitmap with c_1 and c_2) should be further intersected with partition Bitmap before obtaining count. This way, partitions can be created up to any level without changing the original Data set.

3.5.5 Running Decision Trees on GPU

At the beginning of the algorithm, the entire data set will be copied to GPU. But a copy of the Data set will be stored in CPU since it will be referred when partitioning data and creating the split.

The main difference between the CPU and GPU versions is in the places where main computation happens. Taking a similar approach to Naïve Bayes, in this algorithm also, a two dimensional matrix gets initialised. At the point of evaluating a split, this matrix will be transferred to Device memory, and counting kernels will get invoked. In Naïve Bayes, counting only occurs one time. But in Decision Tree counting phase repeats multiple times. Based on the populated contingency table, the Gain Ratio will be calculated and splitting attribute will be determined. To indicate each partition, a new existential Bitmap is created and added as a reference to an Instance (an instance of a class) of the Data set. Since the tree is grown in a Depth first fashion, the partition corresponding to the first attribute value will be used to perform the next series of kernel invocations. But now since the counts are to be obtained using a subset of data, the existential bitmaps will be transferred to the GPU. The kernels would use this existential Bitmap while producing partition-wise counts. Once the contingency table for a partition is filled, all existential bitmap will be removed from the device, but the original Data set would remain until all computations are done.

The same batching modes proposed for Naïve Bayes can be straightaway applied for Decision Trees. Different from Naïve Bayes, for Decision trees, an

existential Bitmap needs to be transferred to memory when obtaining counts for a partition. This frequent data transfer becomes a significant overhead when branching factor increases or when tree grows to many levels. One way to deal with this would be to cache the existential Bitmaps, but which too would exhaust Device memory very quickly if the tree becomes very large. Due to this limitation, speed ups obtained for Decision Trees were modest and improvements were only visible when Data sets were large.

We have only explained about building Tree with Bitmaps, since the approach with Bit Slice is quite similar Bitmap approach. The only noticeable difference between the two representations is that, with Bitmaps the only computing happening is intersection, while in Bit Slices additional processing needing to happen to generate the Bitmap.

In this chapter we explained in detail about the data structures we have used, architecture of the framework and how algorithms are implemented. We took a digression into explain about Bit-Slice processing, since it is fundamental in understanding the novelty of the framework.

Chapter 4

EXPERIMENTAL RESULTS

This section presents the experiments we carried on the framework and results we obtained. Since our main goal was to improve running time of Algorithms, in all experiments execution time was measured as the metric.

Since our primary target is to measure performance gain obtained by Bitmap and Bit-Slice variants on GPUs, results are compared with CPU variants which use those representations. While implementing algorithms, we have been careful only to change the parts where computations are done, since it is the only way to ensure that a fair environment is provided. If we would have compared our results with a different implementation, it would have been with GPUMiner. But since GPUMiner does not support the exact two algorithms we implemented, we could only do the comparison with CPU variants.

At the core of the framework we have several algorithms which process Bitmaps and produce a count, among which Co-OccurrenceCount is the most prominent one. First we present results for Co-OccurrenceCount under different conditions, to prove that the core operation is fast for a variety of conditions. Then we move onto talk about high-level algorithms.

While implementing algorithms, we used the implementation provided by Weka as a reference. The original Weka was written in java. But since our kernels were written in CUDA, we used the C++ port which was available at [22]. For a single algorithm there would be several implementations. A CPU implementation which uses, Weka's in built data structure. Two other CPU implementations which are written using Bitmaps and Bit-Slices. Two GPU implementations which would use Bit-Slices and Bitmaps but would operate in non-batched mode and two implementations which uses Bit-Slices and Bitmaps with batched operations (on GPU). For Naiv̈e Bayes, we also implemented an additional version which uses a simple columnar structure. All these algorithms would have a part

where a count matrix or a contingency table is populated. Between different implementations only part that differs is the part that calculate the contingency table. In GPU implementations, actual computation happen on the GPU and a complete contingency table would be transferred back to CPU.

Accuracy of the algorithms were measured with two approaches.

- By Comparing accuracy of the models built.
- By directly comparing model parameters.

To make sure that each implementation is building the same model, we compare model parameters. To ensure that different iterations of the same implementation creates same model across iterations, we compare accuracy.

4.0.1 Data Sets

For experiments with core algorithms, we are using synthetic Data Sets generated by imputing different properties. For evaluating main High-level algorithms we used Real-world Data Sets available in UCI machine learning repository [23]. To test whether our implementation scales well with increasing volumes, we test our Algorithms by increasing both the number of instances and attributes. By using Data Sets with different attributes we achieve both Attribute-wise and instance-wise scaling.

- USCensus - This is a Categorical Data Set with 68 attributes and two million rows. This is created based on a survey done 1990, and this Data Set has been used for Clustering tasks. Available at [24]
- PokerHand - This Data Set available at [25] contains 11 attributes and 1 million instances. This Data Set consists of both Categorical and Integer data, but we are treating all instances as Categorical. Data Set has been used in Classification tasks.
- KDDCup99 - This Data Set with 42 attributes and four million rows, is based on a variety of intrusions simulated in a military network environment. Data Set contains Categorical and Numeric data. While using for our

experiments we had to convert Numeric columns to Categorical by splitting those to ranges. This is hosted at [26] and has been used in Clustering and Classification Algorithms.

4.0.2 Experimental setup

All experiments were performed on a computer with Intel Core 17-2600 CPU at 3,40GHz, with HyperThreading, 16GB of main memory, and equipped with a GeForce GTX480 graphics card. The GPU consists of 15 SIMD multi-processors, each of which has 32 cores running at 1.4 GHZ. The GPU memory is 1.5 GB with the peak bandwidth of 177 GB/sec.

4.0.3 Results for Co-OccurrenceCount

These were actually a set of preliminary experiments we performed to check if the particular operation is fast enough on GPUs. Since all other high level algorithms are using this step as a building block, we first needed to get an idea how much gain we can get on the individual operation. For this experiment we used a Data Set with only two columns, each with a cardinality of 16. In total there were 256 patterns. Results of the first experiment is shown in Fig. 4.1. Actually the three figures, 4.1, 4.2 and 4.3 depict the results of the same experiment. The plots were created in three segments as the results cannot be shown in the same plot due to the variations in results.

Fig. 4.1 show the scaling characteristics of CPU algorithms. Here we are comparing CPU variants of Bit-Slice and Bitmap algorithms with Standard-CPU. Note that the two variants do show an improvement since those structures are space efficient and consume less space than the normal Row-wise representation.

In Fig. 4.2 the same two Bit-Slice and Bitmap graphs are shown again to make a reference to the CPU variants. The comparison between CPU and GPU variants of the same Algorithm shows the magnitude of speed up offered by GPU. Comparison between GPU variants are shown in 4.3, where the difference between Batched and non-Batched implementations are highlighted.

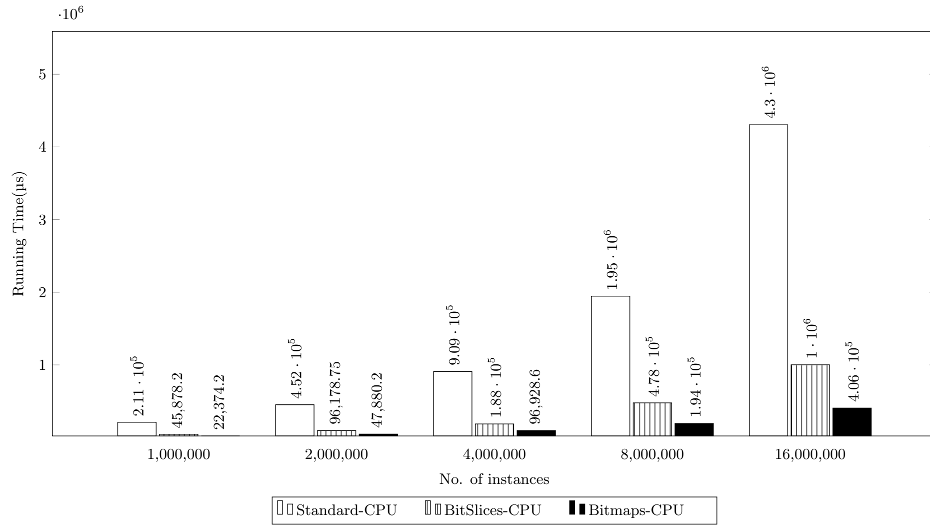

Figure 4.1: Execution time vs no. of instances - CPU Algorithms

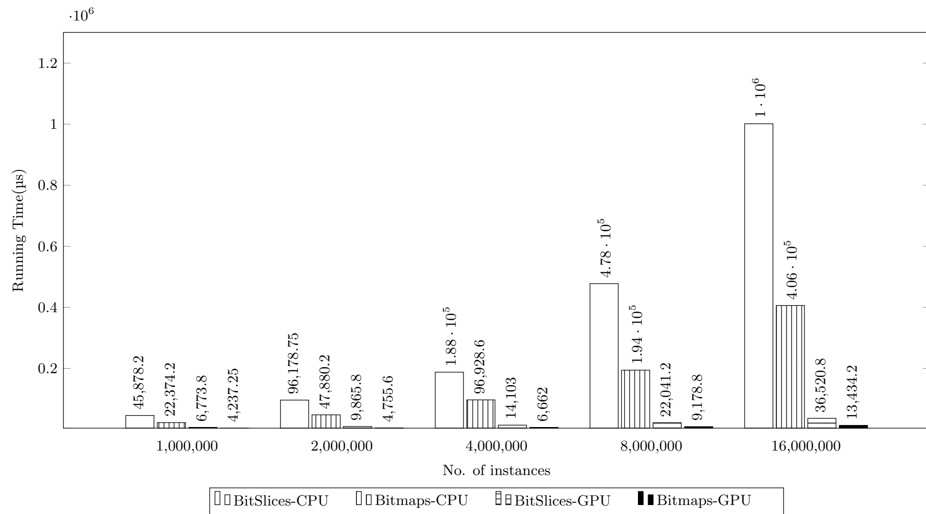

Figure 4.2: Execution time vs no. of instances - CPU and GPU Algorithms

Fig. 4.4 shows how GPU variants behave as number of co-occurrences are increased. We used a Data Set with 16 million elements while changing the cardinality of the two columns. According to the graph, Batched Bitmap variants scales better as patterns are increased.

4.0.4 Running time for Naïve Bayes

The goal of following tests is to assess performance of two Naïve Bayes implementations for GPU, with respect to CPU implementations. We mainly performed

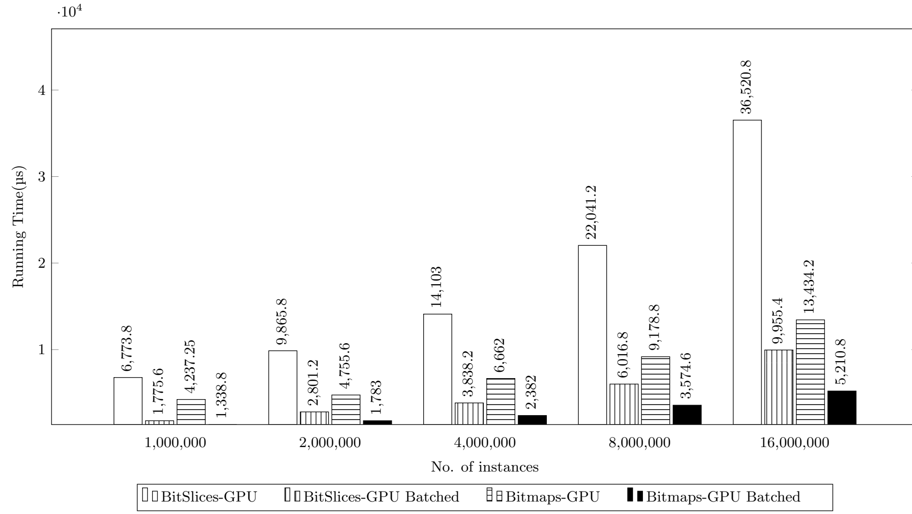

Figure 4.3: Execution time vs no. of instances - GPU Algorithms

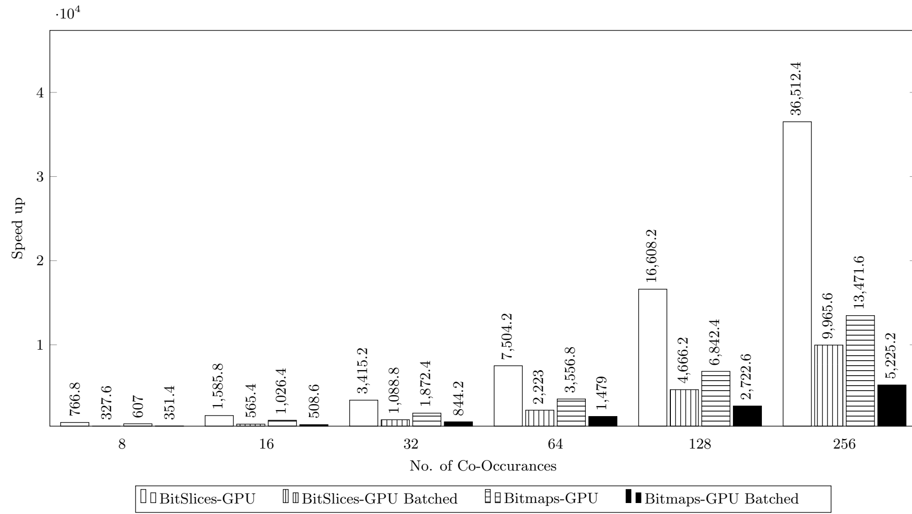

Figure 4.4: Execution time vs no. of Patterns

the test by using different Data Sets and recording execution time of each variant. For the experiments we used 7 different implementations which are summarized in Table 4.1.

In Fig. 4.5 we show the comparison with different Data Sets, according to which we can see a clear difference between CPU and GPU variants. Here we have directly compared CPU implementation with Batched-GPU variants, because that would give the maximum difference. While measuring time, we only measured the time taken to build the model. Transfer times were excluded because a transfer would be done only once for multiple executions and can be

Table 4.1: Different implementations and their descriptions.

Algorithm	Description
Standard-CPU	Unmodified implementation provided by Weka.
Bit-Slice-CPU	Algorithm that uses Bit-Slices, which runs on the CPU.
Bitmap-CPU	An implementation running on CPU using a Bitmap representation.
Bit-Slices GPU	The variant using Bit-Slices, which runs on GPU.
Bit-Slices GPU Batched	The same variant as above, which performs operations in batches.
Bitmap GPU	An implementation running on GPU which uses Bitmaps.
Bitmap GPU Batched	The bitmap variant running on GPU running operations in batches.

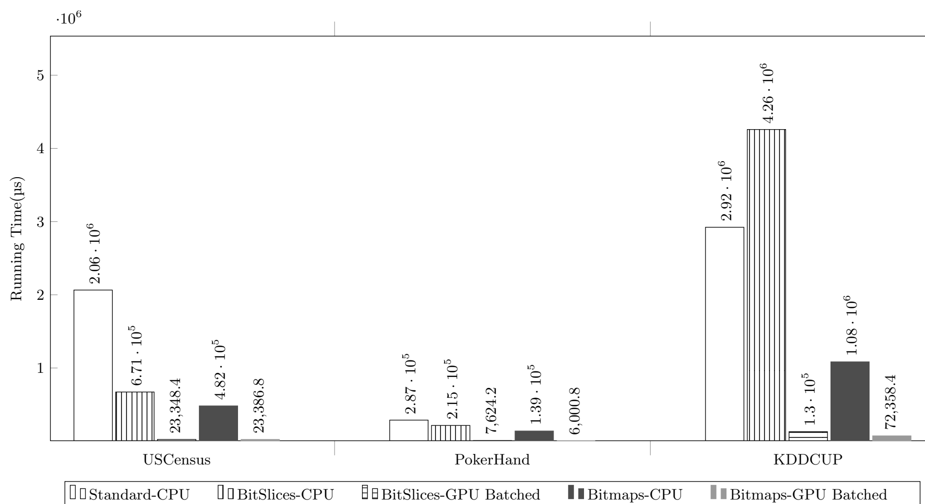

Figure 4.5: Execution times with Different Data sets Results for Naïve Bayes.

considered as a one time operation. And as larger Data Sets are used, transfer time accounts for the larger proportion of time, which hides the speedup obtained by GPU. The graphs show an average time which is an average of six iterations. In the Standard-CPU variant, data is held in a row-wise structure and it does not perform any encoding. Both in Bit-Slice and Bitmap representations, a reduction in Data Volume can be observed since the size of the column depends on cardinality of the attributes. Reason for providing CPU variants for Bitmap and Bit-Slice versions is to isolate this impact. When comparing CPU variant with the GPU variant we can eliminate the boost given by specific characteristics

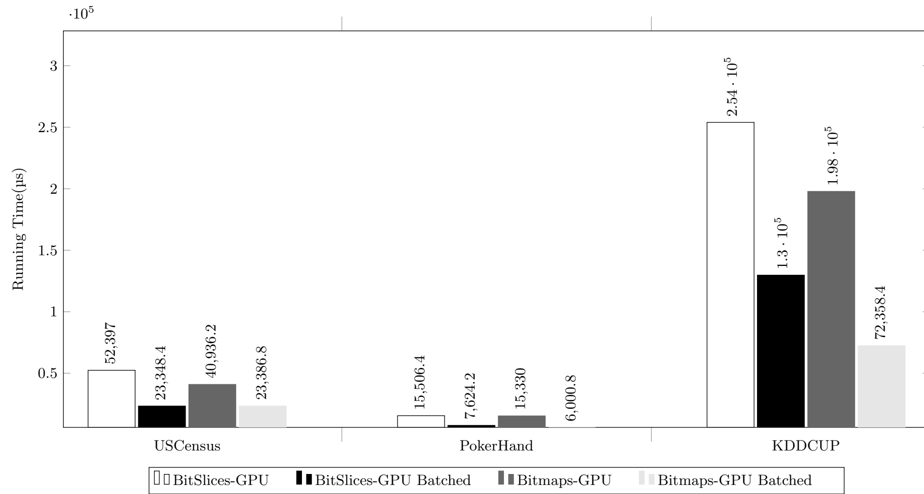

Figure 4.6: Executions on GPU with the three Data Sets.

of the representation and focus on speed up solely given by parallel processing. The two Data Sets USCensus and PokerHand exhibit a similar pattern where Standard-CPU takes the longest time. In KDDCUP, Bit-Slices-CPU accounts for the slowest execution. While investigating on this what we found was that, column cardinality of this Data Set was higher compared to the other two, and now algorithm needs to run more iterations to populate the contingency table. Batched execution is resilient to this, since iterations do not linearly increase as cardinality increases.

When obtaining time for GPU based algorithms, initial result was omitted as it usually takes an abnormally longer time since it also includes the time to load CUDA drivers into runtime and do platform specific operations.

Fig. 4.6 shows a clear comparison between GPU variants. It shows running time for the same Bit-Slice and Bitmap implementations, but additionally it also shows results for non-batched mode. For USCensus and PokerHand Data Sets, there is a noticeable difference between the two non-batched variants. Bitmap algorithm runs faster than the Bit-Slice one. But for the same Data Sets, there is very little difference between batched variants. While looking into the Data Sets we found that, in both, there are many attributes with a cardinality less than 4. In the non-batched mode, Bitmap based algorithm would read two bitmaps

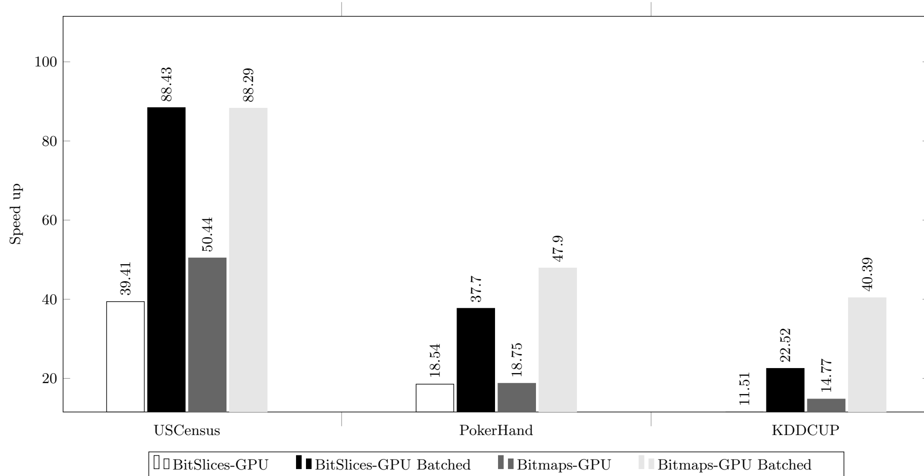

Figure 4.7: Speedup over Standard-CPU on different Data Sets.

to produce result of a single intersection, but to produce the result for the same intersection, Bit-Slice algorithm would read 4 Bit-Slices. In batched mode, both the variants will be reading 4 Bitmaps to produce 4 results. Since memory transfers are more uniform in batched mode, Bit-Slice and Bitmap algorithms run equally fast. For KDDCup, the case is different, because most of its attributes were numerical and while converting to categorical attributes, purposefully the ranges were split to have many different categories. However for KDDCUP, we get Batched Bitmap version running nearly two times fast as the Batched Bit-Slice variant. This gives an indication that, for processing Data Sets with higher cardinalities Bitmap variant suites better, but again this needs to be confirmed through careful analysis, since Bitmaps have a tendency to increase volume as cardinality increases, which again can bring down performance. However, since these variants can be easily switched, we have provided the option to select the most appropriate representation based on the characteristics of the Data Set.

In all cases, there is a clear difference between batched and non-batched variants. A batched variant reports nearly the half the execution time of the non-batched counterpart. In some cases, Bitmap based algorithms have performed better than Bit-Slice based ones, which is due to the latter having to read more bits to produce the same result.

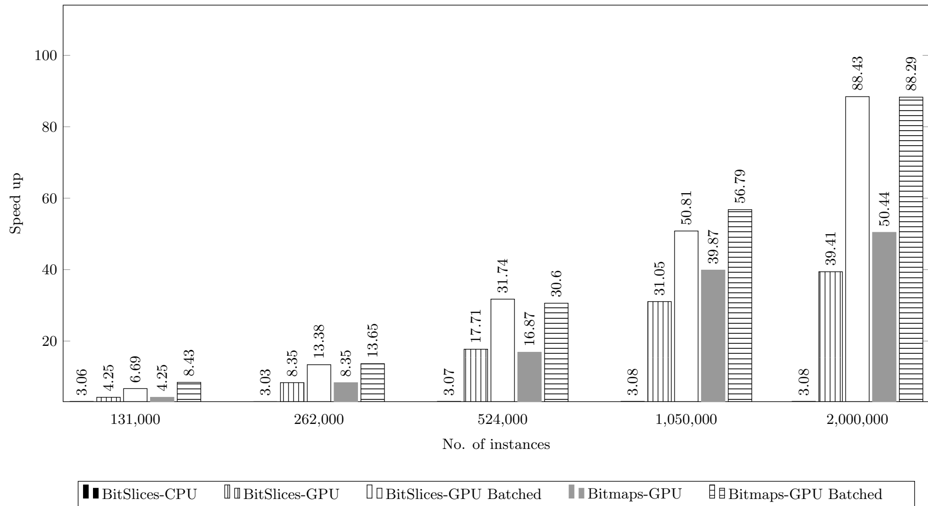

Figure 4.8: Naïve Bayes speedup vs instance count

Fig. 4.7 Shows Speedups obtained against Standard-CPU execution. For USCensus Data Set, the maximum speed up is nearly 80.

We also wanted to see how speed up changes as the data volume increases which is depicted in Fig. 4.8. We measured running time for each variant as we changed the instance count. To obtain the speed up we used running time of Standard-CPU variant as the reference point. In this experiment we only used USCensus Data Set. Bit-Slice-CPU variant was plotted to show the differences between CPU and GPU variants. Unlike the GPU variants, Bit-Slice-CPU is showing a constant speed up across all Data Set sizes. Another interesting property we can see is that, even though there is a noticeable difference between Bit-Slice and Bitmap GPU variants, the Batched counterparts show essentially similar speed ups. This can be due to the characteristics of the particular Data Set, but again this suggests that when using Batched mode Bit-Slices can used as a generic representation hence it performs equally well as Bitmaps.

4.0.5 Results for Decision Trees

Fig. 4.9 shows the results for Decision Tree algorithm. Same evaluation and preparation steps followed for Naïve Bayes were used while running the experiments and verifying model accuracy. However, we did not execute non-batched

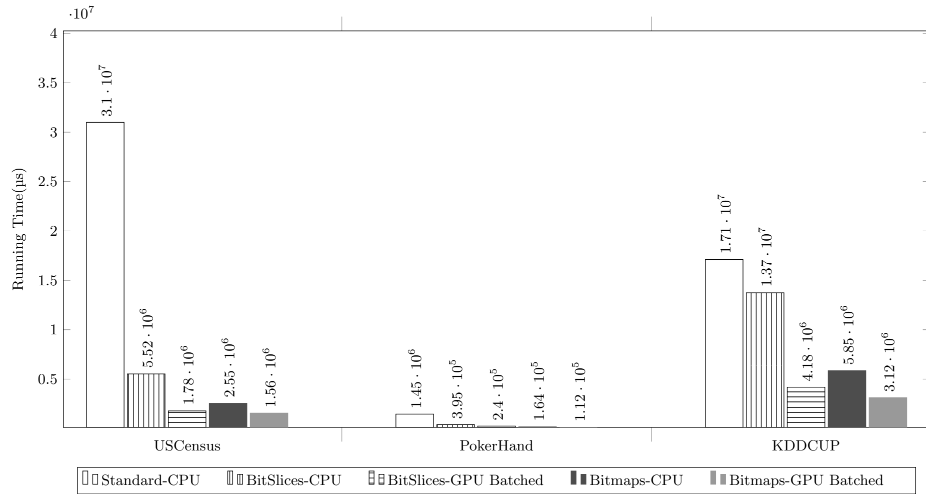

Figure 4.9: Execution times for Decision Tree with Different Data Sets

modes for Decision Trees, since batched modes themselves were not giving a considerable speed up. There is a main difference between Naïve Bayes and this, which is counting phase occurring multiple times for this algorithm. Each new counting happens on a different partition and to filter counts by the partitions we have to maintain an existential Bitmap. With the approach we are following, these get generated on the host and then get copied over to Device. Each time a new partition is to be processed, this existential bitmap gets transferred from host to device, which becomes the main bottleneck in achieving higher speed ups. Still the GPU variants finish faster than CPU ones, but the speedups are small.

Fig. 4.10 shows speed ups reported against Standard-CPU variant. Since we only implemented batched variant on GPU, we are only showing speedup against two algorithms.

In this chapter we discussed about the experiments we performed and the results obtained. Through our experiments, we show that the techniques we use, speeds up algorithm executions by a significant factor. Going by the previous studies, we show results with multiple Datasets. We also show that our GPU algorithms scales better in terms of data volume and number of patterns.

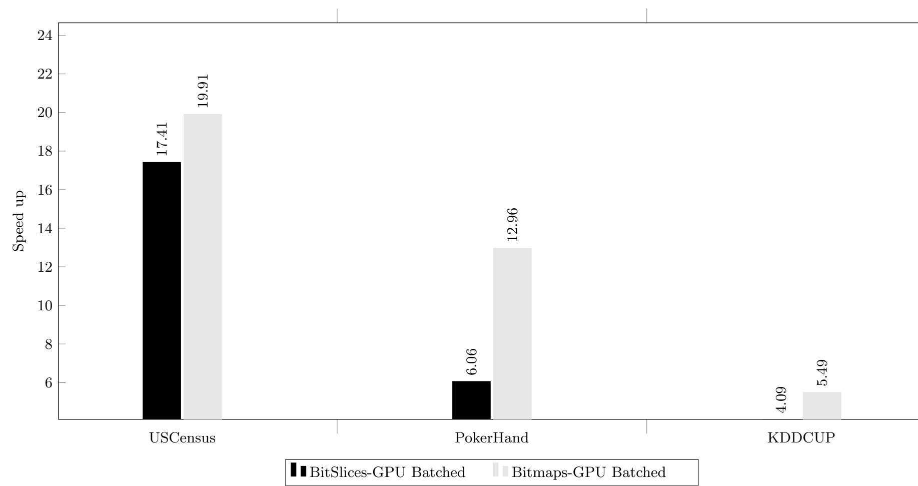

Figure 4.10: Speedup over Standard-CPU on different Data Sets.

Chapter 5

CONCLUSIONS AND RECOMMENDATIONS

According to the experiment results we showed in the previous section, we have been able to achieve a speed up over 75 for Naïve Bayes and speed up over 12 for Decision Trees. This is by keeping a uniform Data store and without implementing algorithm specific structures. Through these experiments we show that if a Data Mining algorithm can be expressed as a series of counting operations done on a Bitmap Data Structure, then those can be implemented on our framework to obtain a good speed up. Even though we have been mainly talking about counting operation, it is not the only operation we can speed up with Bitmap structures. Columnar operations such as RangeCount, Sum, Squared Sum are some other operations which can be implemented and sped up with Bitmap processing.

Our broader intention was to extend Bitmap processing which currently is mostly used in Frequent Itemset Mining. By,

- Implementing a Columnar Data Store based on Bitmaps
- Providing a set of GPU Kernels to manipulate the Data Store
- Implementing two Data Mining Algorithms

we have shown that algorithms which can be expressed in counting (more generally columnar) operations can be executed on a Bitmap based framework. Since the framework combines Parallel processing with the advantages offered by Bitmaps, we have shown that significant speed ups can be obtained by implementing algorithms on this framework.

For the current scope of work, we only considered nominal attributes and integers. But in order to support a wider variety of Applications we need the framework to support numerical attributes as well. This can be done as a future work.

Further, there are different ways of optimising core Bitmap operations. Using Streams, changing task allocation and using vectorised Data Types for storing data are few approaches we can take. Another aspect we need to improve is limiting Bitmap transfers in Algorithms like Decision Trees.

Through this study we have shown how Counting based Algorithms can be fitted into a common Bitmap based framework. We believe that the work covered in this study would provide a ground work for building a Generic Bitmap based Data Mining framework.

References

[1] *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, 3 edition, 2011.

[2] Bernard Marr. How much data do we create every day? the mind-blowing stats everyone should read. URL: https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/#5e3bdedf60ba, 2018. Online; Accessed 16 March 2019.

[3] Apache hadoop. URL: https://hadoop.apache.org/, 2018. Online; Accessed 16 March 2019.

[4] Apache spark™- unified analytics engine for big data. URL: https://spark.apache.org/, 2018. Online; Accessed 16 March 2019.

[5] A. Gainaru and E. Slusanschi. Framework for mapping data mining applications on gpus. In *2011 10th International Symposium on Parallel and Distributed Computing*, pages 71–78, July 2011.

[6] Wenbin Fang, Ka Keung Lau, Mian Lu, Xiangye Xiao, Chi Kit Lam, Philip Yang Yang, Bingsheng He, Qiong Luo, Pedro V. S, and Ke Yang. Parallel data mining on graphics processors. Technical report, 2008.

[7] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. A performance study of general-purpose applications on graphics processors using cuda. *Journal of Parallel and Distributed Computing*, 68(10):1370 – 1380, 2008. General-Purpose Processing using Graphics Processing Units.

[8] C. Silvestri and S. Orlando. gpudci: Exploiting gpus in frequent itemset mining. In *2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 416–425, Feb 2012.

[9] Kang-Wook Chon, Sang-Hyun Hwang, and Min-Soo Kim. Gminer: A fast gpu-based frequent itemset mining method for large-scale data. *Information Sciences*, 439-440:19 – 38, 2018.

[10] Wenbin Fang, Mian Lu, Xiangye Xiao, Bingsheng He, and Qiong Luo. Frequent itemset mining on graphics processors. In *Proceedings of the Fifth International Workshop on Data Management on New Hardware*, DaMoN '09, pages 34–42, New York, NY, USA, 2009. ACM.

[11] Ferenc Bodon. A trie-based apriori implementation for mining frequent item sequences. In *Proceedings of the 1st International Workshop on Open Source Data Mining: Frequent Pattern Mining Implementations*, OSDM '05, pages 56–65, New York, NY, USA, 2005. ACM.

[12] Christian Böhm, Robert Noll, Claudia Plant, Bianca Wackersreuther, and Andrew Zherdin. *Data Mining Using Graphics Processing Units*, pages 63–90. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[13] Liheng Jian, Cheng Wang, Ying Liu, Shenshen Liang, Weidong Yi, and Yong Shi. Parallel data mining techniques on graphics processing unit with compute unified device architecture (cuda). *J. Supercomput.*, 64(3):942–967, June 2013.

[14] M. Harris. Optimizing parallel reduction in cuda. URL: `https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf`, 4 2016. Online; Accessed 16-Feb-2019.

[15] G. Andrade, F. Viegas, G. S. Ramos, J. Almeida, L. Rocha, M. GonÃğalves, and R. Ferreira. Gpu-nb: A fast cuda-based implementation of naÃŕve bayes. In *2013 25th International Symposium on Computer Architecture and High Performance Computing*, pages 168–175, Oct 2013.

[16] John C. Shafer, Rakesh Agrawal, and Manish Mehta. Sprint: A scalable parallel classifier for data mining. In *Proceedings of the 22th International Conference on Very Large Data Bases*, VLDB '96, pages 544–555, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[17] Manish Mehta, Rakesh Agrawal, and Jorma Rissanen. Sliq: A fast scalable classifier for data mining. In Peter Apers, Mokrane Bouzeghoub, and Georges Gardarin, editors, *Advances in Database Technology — EDBT '96*, pages 18–32, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[18] Yisheng Liao, Alex Rubinsteyn, Russell Power, and Jinyang Li. Learning random forests on the gpu. 12 2013.

[19] Patrick O'Neil and Dallan Quass. Improved query performance with variant indexes. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, SIGMOD '97, pages 38–49, New York, NY, USA, 1997. ACM.

[20] Ian H. Witten, Eibe Frank, and Mark A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2011.

[21] Cécile Favre and Fadila Bentayeb. Bitmap index-based decision trees. In *Proceedings of the 15th International Conference on Foundations of Intelligent Systems*, ISMIS'05, pages 65–73, Berlin, Heidelberg, 2005. Springer-Verlag.

[22] Weka-c++. URL: `https://sourceforge.net/p/wekacpp/`, 2007. Online; Accessed 16 March 2019.

[23] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

[24] Us census data (1990) data set. URL: `https://archive.ics.uci.edu/ml/datasets/US+Census+Data+(1990)`, 1990. Online; Accessed 16 March 2019.

[25] Poker hand data set. URL: `https://archive.ics.uci.edu/ml/datasets/Poker+Hand`, 1990. Online; Accessed 16 March 2019.

[26] Kdd cup (1999). kdd cup 99 intrusion detection datasets. URL: http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html, 1999. Online; Accessed 16 March 2019.